

AD-A103 772

SPECIFICATION AND DESIGN METHODOLOGIES FOR HIGH-SPEED
FAULT-TOLERANT ARRA. (U) CALIFORNIA UNIV LOS ANGELES
DEPT OF COMPUTER SCIENCE M D ERCEGOVAC ET AL. JUN 87

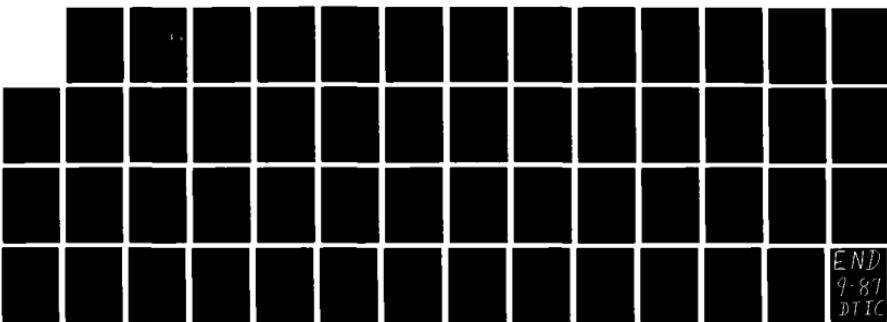
1/1

UNCLASSIFIED

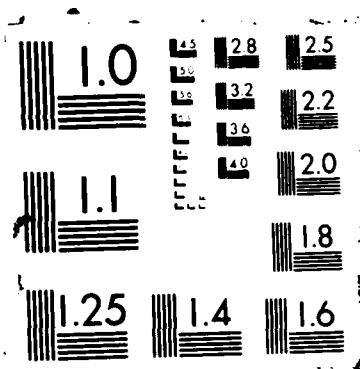
NO0014-83-K-0493

F/G 9/1

NL



END
9-87
DTIC



FINAL REPORT

SPECIFICATION AND DESIGN METHODOLOGIES FOR
HIGH-SPEED FAULT-TOLERANT ARRAY ALGORITHMS AND STRUCTURES
FOR VLSI

Office of Naval Research
Contract No. N00014-83-K-0493

Principal Investigator

Miloš D. Ercegovac

Co-Principal Ivestigator

Algirdas Avizienis

Faculty Associate:

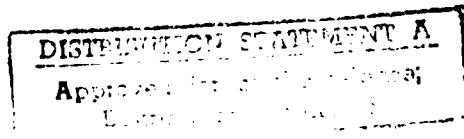
Tomas Lang

AD-A183 772

UCLA Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024
(213) 825-2660

DTIC
ELECTED
S D
AUG 04 1987
CSD

June 1987



87 7 14 063

Table of Contents

1. Summary of the Project Objectives	3
2. Summary of Contributions: Task 1	4
2.1 Introduction	4
2.2 Specification of Hardware Functions and Algorithms in vFP	4
2.3 Obtaining Layouts from FP	6
2.4 Evaluation of Designs	7
2.5 Interfacing vFP Design System with Existing VLSI CAD Tools	7
2.6 Compiler Research and Developments	7
2.7 Algorithms for VLSI Implementation	8
2.8 Future Research	8
3. Summary of Contributions: Task 2	10
4. Publications Resulting from This Project	12

Appendices: Selected Publications

1. D.R. Patel, M. Schlag, and M.D. Ercegovac,
"vFP: An Environment for the Multi-Level
Specification, Analysis, and Synthesis
of Hardware Algorithms"
2. F. Meshkinpour and M.D. Ercegovac,
"A Functional Language for Description and
Design of Digital Systems: Sequential Constructs"
3. A. Avižienis, "Arithmetic Algorithms for
Operands Encoded in Two-Dimensional Low-Cost
Arithmetic Error Codes"
4. M.D.F. Schlag, "Layout from a Topological Description"
(Abstract)*
5. J. Moreno, "A Proposal for the Systematic Design
of Arrays for Matrix Computations"
(Abstract)*

*Reports will be submitted upon request

1. SUMMARY OF THE PROJECT OBJECTIVES

For convenience we summarize here the project objectives as stated in the research proposal. This research in the methodologies for the specification and design of high-speed, fault-tolerant VLSI array structures has two related objectives (1) a high-level language approach to the specification and simulation of VLSI algorithms and networks using a functional-style (LISP-like) language (Task 1), and (2) cost-effective methods to introduce fault-tolerance (error detection, fault location, retry, and reconfiguration) into VLSI-implemented systolic systems and similar computing arrays (Task 2).

Task 1: Functional Language Approach to VLSI CAD+

Principal Investigator: Milos D. Ercegovac

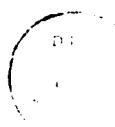
The major goals are the development and implementation of a functional-style language for specification of VLSI structures which allows multilevel simulation, performance analysis, algebraic transformation techniques, and layout planning. The proposed high-level functional-style language approach provides a clean separation of functional and structural specifications, and supports strongly the multi-level, hierarchical design; the language is executable at any level of abstraction to allow for early evaluation and checking of designs; it has an efficient and comprehensive built-in performance evaluation mechanism which allows a selective performance observation; and it supports a semi-automatic design methodology under implementation constraints and system requirements.

Task 2: Fault-Tolerance in VLSI Systolic Arrays,

Principal Investigator: Algirdas Avižienis

A systolic VLSI system consists of a set of interconnected cells, and information between the cells flows in a pipelined fashion. To provide fault-tolerance for a systolic system, the most fundamental requirement is to provide an effective and low-cost method for the immediate detection of errors that occur in the numerical information that is generated by the cells and forwarded to other cells or to I/O ports. The occurrence of transient malfunctions and the complexity of structure of a VLSI systolic system rules out periodically applied diagnostic tests as an effective fault detection method. The remaining approach is to provide concurrent error detection that takes place side-by-side with regular computation whenever the systolic system is carrying out its activity.

+ This research was also supported in part by the State of California MICRO Program



2. SUMMARY OF CONTRIBUTIONS: Task 1

This task deals with a study and development of a high-level language approach in specification, simulation, performance evaluation and chip layout planning for VLSI digital systems. A high-level applicative (functional) language, implemented at UCLA, allows combining of top-down techniques of functional and structural specification of systems with bottom-up specification of implementation constraints such as the size of circuit layout and wiring patterns. It also provides highly modular algebraic specification of digital systems suitable for formal transformations, simulation, and a powerful method of topological interpretation which generates diagrams at any level of abstraction. Several versions of the language, the simulation and performance evaluation tools and a graphics interface have been developed and implemented on the DEC VAX 11/750 under the UNIX operating system.

2.1 Introduction

The complexity of VLSI requires the application of CAD tools at all levels of the design process. In order to be effective, these tools must be adaptive to the specific design. In this project we studied a design method based on the use of applicative languages [Bac78] for the specification, evaluation and synthesis of hardware algorithms. A functional language for specification of hardware systems is attractive because it provides both a behavioral and structural information about a circuit implementing the system [Lah81, Joh84, Mes84, Pat85, She84, Sch86, Wor86]. As a consequence, a behavioral specification implies a topology of the circuit which allows generation of "abstract" layouts. These layouts are refined by introducing geometrical constraints to produce physical layouts.

Our methodology is supported by a set of tools developed at UCLA. The goal of the system is to provide designers with an environment in which they can rapidly explore various alternative designs. Thus it is possible to specify the algorithm at any level of abstraction and have the system rapidly evaluate certain parameters (e.g., delays) and provide feedback in the form of automatically generated floorplans. The advantage of using an applicative language is that it ties together the specification of the algorithm, the synthesis of the circuit, and the evaluation of the implementation. The algebraic basis of FP allows formal transformations of the specification to improve the layout without changing its function.

2.2 Specification of Hardware Functions and Algorithms in vFP

A program in a functional language is a function that maps objects into objects. Objects are either atoms (numbers or strings) or sequences of objects. There is a special atom ? denoting an undefined value. Any sequence that contains ? is undefined. The language includes primitive functions, functional (combining) forms, and means of defining functions. A computation is invoked by applying the function to an object. There are no variables and all FP programs are generic, i.e., independent of the size of their arguments.

The FP language we use is based on Backus' FP [Bac78] with the following additions: parameters to function definitions are allowed; both the infix and prefix modes for the arithmetic, logical, and predicate functions can be used; there are additional primitives and functional forms; and extensions for the specification of sequential systems are introduced [Mes84, Mes85, Pat85, Sch86, Wor86]. The primitive functions map objects to objects. They include

arithmetic	$+:<1 5> \rightarrow 6$	$*:<2 5> \rightarrow 10$
logical	$andg:<1 0> \rightarrow 0$	$org:<0 0> \rightarrow 0$
predicate	$atom:<a b c> \rightarrow F$	$=:<12 12> \rightarrow T$
selector	$2:<1 a 3 5 b> \rightarrow a$	$last:<4 3 2 1> \rightarrow 1$

and structure modifying functions such as

transpose	$trans: <<1 2 3><4 5 6>> \rightarrow <<1 4><2 5><3 6>>$
append left	$apndl: <a <b c d>> \rightarrow <a b c d>$
distribute right	$distr: <<1 2 3>4> \rightarrow <<1 4><2 4><3 4>>$

Functional forms map functions or objects to functions. For example,

compose	$f@g:x \rightarrow f:<g:x>$
construct	$[f,g,h]:x \rightarrow <f:x g:x h:x>$
apply to all	$&f:<a b c> \rightarrow <f:a f:b f:c>$
constant	$%k:x \rightarrow k \text{ if } x \text{ is not ?}$
right insert	$!f:<a b c d> \rightarrow f:<a !f:<b c d>>$

A computation in a digital system consists of moving and transforming data according to some precedence relation. The language provides explicit means for specifying precedences and concurrency (@ and [] functional forms, for example), computational functions (e.g., logical primitives), and routing functions (e.g., selectors). Since a unit of information represented by an atom depends on the level of abstraction, hierarchical specifications are natural. Therefore, FP is suited for describing hardware functions and algorithms.

For example, FP specifications for the following primitive functions used in the design of a carry-save array multiplier are

```

/* FA*: <<a b><y x> → <<c x>s> where 2c+s=a+b+yx */
defun FA*
  [[org@[1,1@2],3],2@2]
  @[1@1,HADD@[2@1,2],3]
  @[HADD@1, andg@2,2@2]
enddef

/* HA*: <<a><y x>> → <<c x>s> where 2c+s=a+yx */
defun HA*

```

```

[[1@1,2],2@1]@[HADD@[1@1, andg@2],2@2]
enddef

/* FA**:<<a b><x y>> → <<c s> where 2c+s=a+b+yx */ 
defun FA**
    [org@[1,1@2],2@2]
    @[1@1, HADD@[2@1,2]]
    @[HADD@1, andg@2]
enddef

/* FA:<<a b.c. → <<c s> where 2c+s=a+b+c */ 
defun FA
    [org@[1,1@2],2@2]
    @[1@1,HADD@[2@1,2]]
    @[HADD@1,2]
enddef

/* HADD:<a b> → <<c s> where 2c+s=a+b */ 
defun HADD
    [andg,org]
enddef

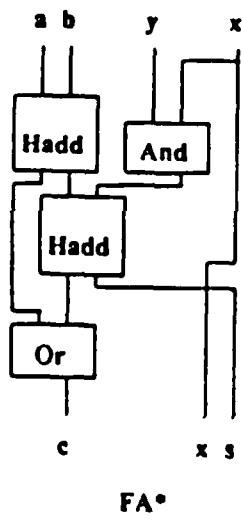
```

By executing symbolically these specifications, it is possible to extract the corresponding topological structure and produce the sketches of functions **FA***, **HA*** and **FA**** as shown in Figure 1. Obtaining layouts from FP expressions is discussed in Section 2.4.

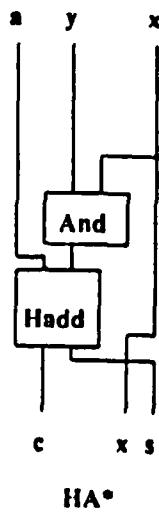
There is no concept of state in an FP program and, consequently, there is no history of execution. All information needed by a computation must be specified as the input to the corresponding function. A sequential system could be described by a function which passes its state as an argument back to itself. This, however, makes symbolic execution of such a FP specification and extraction of its topological structure difficult [Sch86]. Our approach is to describe sequential circuits using the space-time duality. That is, a sequential circuit is described as the folding of a combinational circuit so that the same structure performs a computation in time rather than space [Pat85].

2.3 Obtaining layouts from vFP

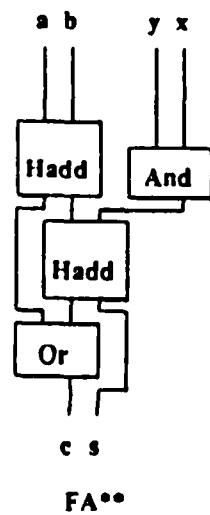
As mentioned in the introduction, a key idea of our design methodology is to deduce the geometry of the layout from a behavioral specification of the circuit rather than to specify the geometry as a part of the behavioral specification. This is possible since an FP program as a behavioral specification of a circuit implies the topology of its organization, i.e., relative positions of the components and their connections in the plane. Schlag [Sch86] has developed a methodology for obtaining layouts from FP expressions. This methodology is based on a formal notion of the planar topology of a circuit, a mapping from FP expressions to planar circuits, and



FA^*



HA^*



FA^{**}

Figure 1. The functions FA^* , HA^* and FA^{**}

a technique for transforming the planar topology of a circuit described in FP into a physical layout.

The following example of a carry-save array multiplier, developed by Schlag [Sch86], illustrates the layout obtained by applying our design methodology. Figure 2 shows a sketch of the function **Mult** with the functions **HA***, **FA***, and **FA**** represented as components. Figure 3 shows the same function with those components expanded. Finally, the completed layout is given in Figure 4. Power and ground wiring has been added to the layout using a graphics editor. Since the specification is generic, multipliers of different precision can be obtained by applying **Mult** function to operands of a desired precision.

2.4 Evaluation of Designs

Sausville has implemented a package that allows timing analysis of circuits represented by FP programs [Sausville 1986]. His work is restricted to uniform delays. Currently a work is under way to extend this package to deal with arbitrary (user-defined) delays.

2.5 Interfacing FP Design System and Existing VLSI CAD

A VLSI design system has been developed using UCLA FP as the specification language by J. Worley [Wor86] and BDL (Block Design Language) [Slu84] as the input language for VLSI CAD tools available at Hewlett-Packard. The circuit synthesis proceeds in three steps: (1) the functional (executable) specification of a digital system is developed and tested, (2) a specific implementation and its net list is obtained by tracing the symbolic execution of the specification, and (3) the trace is processed by trace filters to obtain various design information. For example, there are trace filters to print net lists, count modules and connections, and translate into other design languages. At present, there are translators for *esim*, a switch-level simulator, the circuit level simulator SPICE, and the BDL block description language [Slu84]. A BDL description is then used to drive an actual circuit layout generator. In this case, the user control of the topological features in the layout was traded for utilities provided by an available tool.

2.6 Compiler Research and Developments

A compiler development [Ara86] offers a performance enhancement for execution environment of our FP language. Several important techniques to reduce the run-time load have been introduced and implemented. An efficient and fast threaded FP interpreter/compiler has been implemented [Pun86]. Alkalaj has introduced a very efficient scheme for garbage collection for FP programs executed on a uniprocessor [Alk86, Alk87]. These language processing schemes and tools are essential in building an efficient design environment.

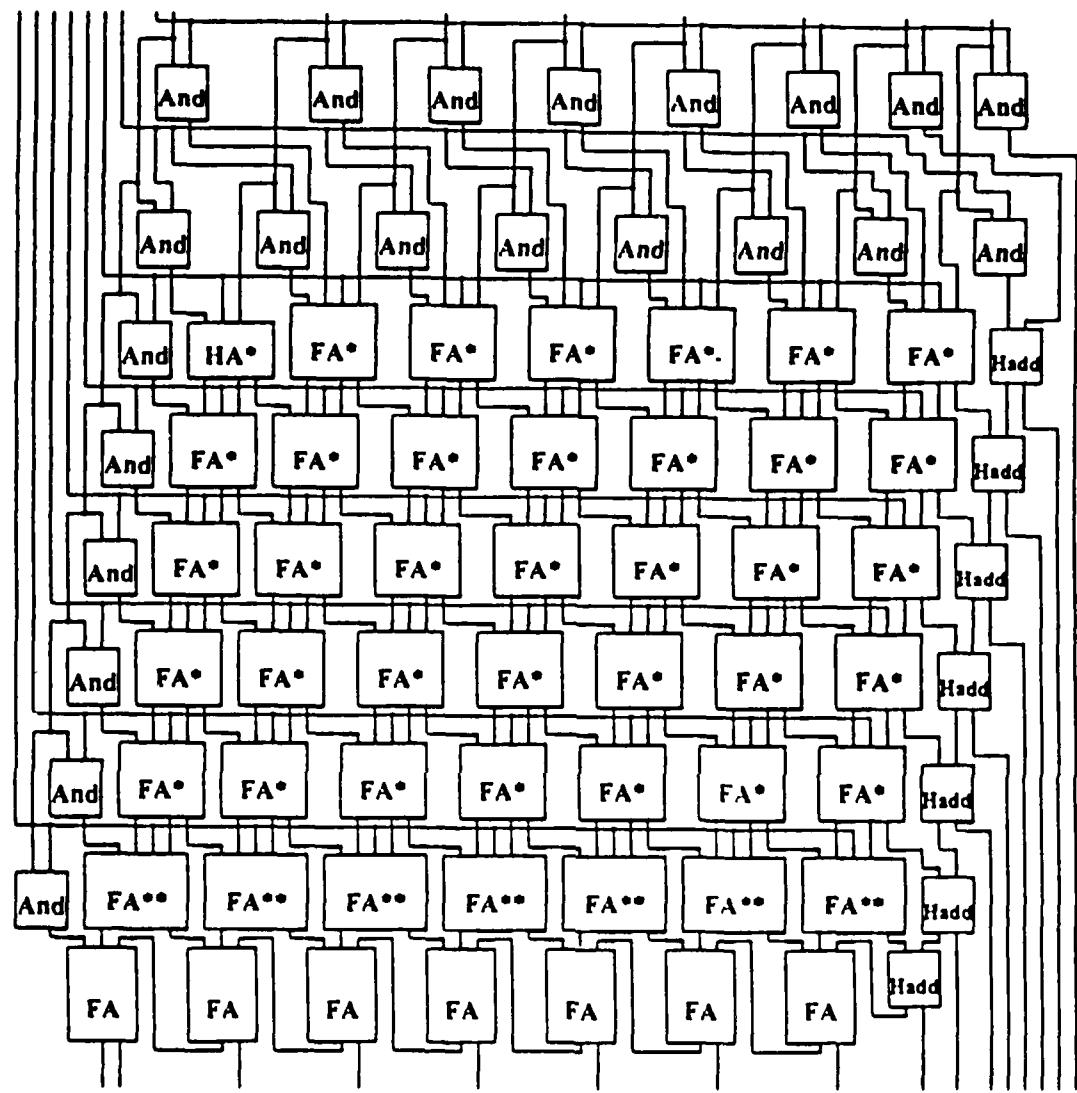


Figure 2. The sketch of the function MULT at a high level

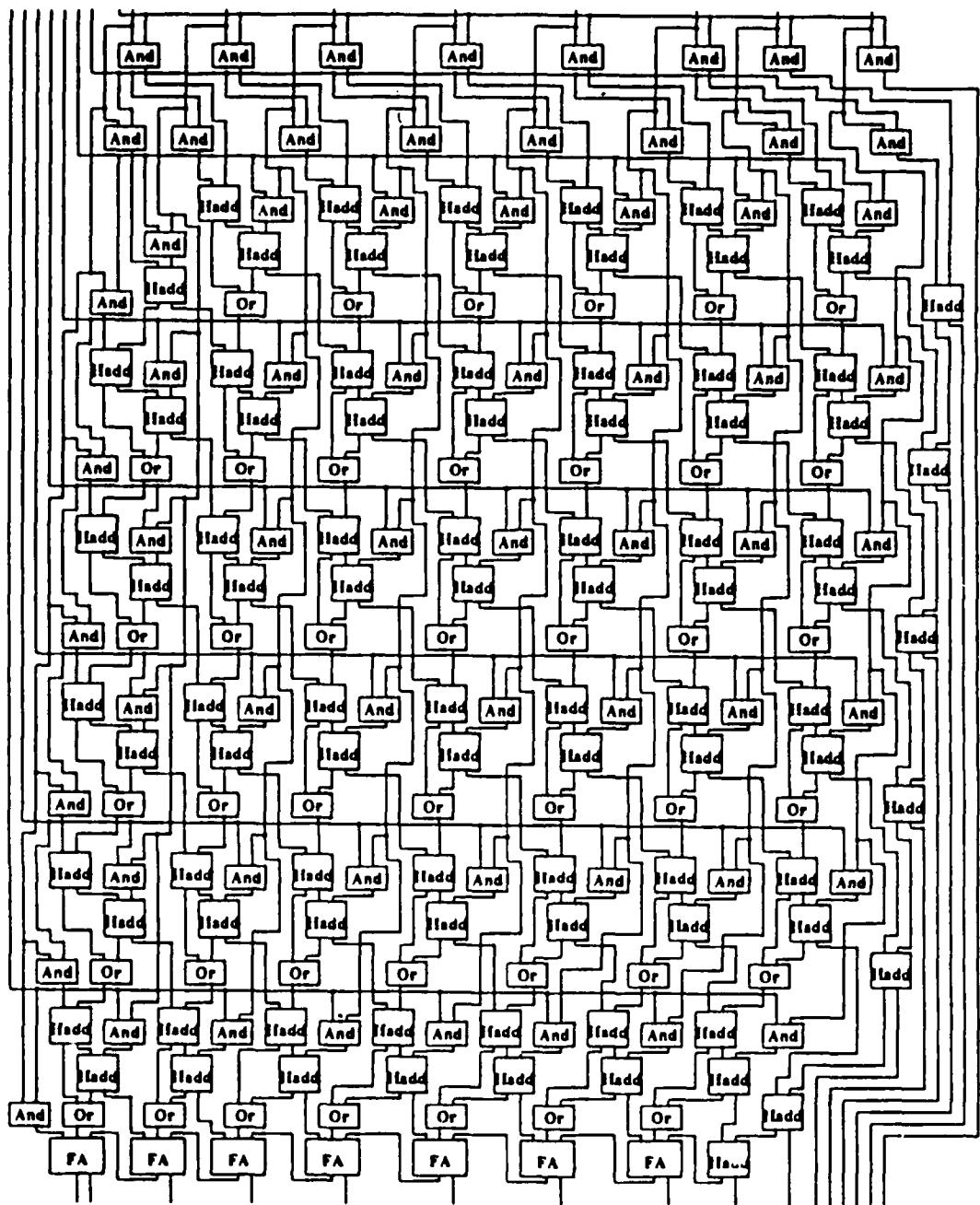


Figure 3. MULT with HA*, FA*, and FA** expanded

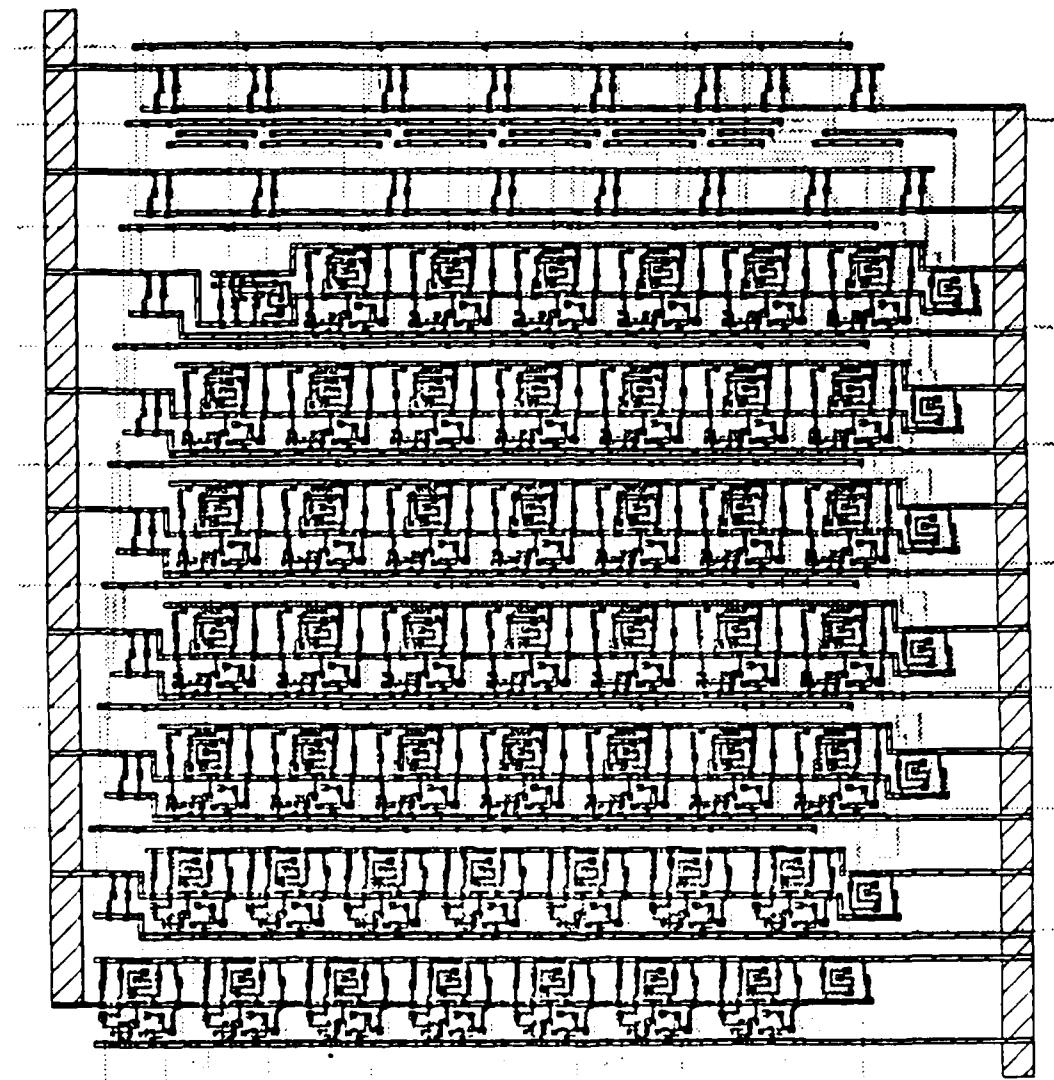


Figure 4. The layout of MULT

2.7 Algorithms for VLSI Implementation

In the area of algorithms for systolic arrays the research focused on analysis of design alternatives and development of algorithms for linear algebra processors. A comprehensive study of alternatives for a singular value decomposition processor has been done [Mor85]. This work has been recently extended into a proposal for the systematic design of arrays for matrix computations [Mor87]. An efficient division algorithm, based on the work Ercegovac and Lang [Erc85], has been developed [Tu86]. In order to provide a flexible and powerfull simulation environment for this type of research, a two-step simulator has been developed. In the first step, an FP specification of the algorithm to be simulated, is symbolically interpreted to produce a corresponding network at the level of given primitives. In the second step this network is used to execute the algorithm and collect statistics.

2.8 Future Research

We are continuing work on two aspects of our FP-based VLSI design system. To utilize well-developed tools available at the lower levels of VLSI circuit design, we are developing an interface between UCLA FP and such tools. VIVID, an integrated VLSI design system, developed at the MCNC, is selected as the target. A translator from UCLA FP into ABCD, a specification language of VIVID, is under development [Wu87]. The second aspect of our continuing research deals with refinement and formalization of the proposed treatment of sequential circuits in UCLA FP [Pat86].

Acknowledgements

The contributions by Leon Alkalaj, Jose Arabe, Farshad Meshkinpour, Jaime Moreno, Dorab Patel, Surapol Pungsornruk, Mark Sausville, Martine Schlag, Paul Tu, John Worley, and Winthrop Wu, who carried out parts of this research and development, are gratefully acknowledged. June Myers and Marilyn Kell provided efficient and friendly administrative and secretarial help.

References

- [Bac78] J. Backus, "Can programming be liberated from the von eumann style? A functional style and its algebra of programs", *Comm. ACM*, Vol. 21, No. 8, August 1978, pp. 613-641.
- [Lah81] D.O.Lahti, "Applications of a Functional Programming Language", MS Thesis, UCLA Computer Science Department, Technical Report CSD-810403, 1981.
- [Pat83] D.R. Patel and M.D. Ercegovac, "A High-Level Language for Hardware-Oriented Algorithm Design", presented at the Symposium "Electronics and Communications in the 80's", Indian Institute of Technology, Bombay, February 1983.

[Mes84] F. Meshkinpour, "On Specification and Design of Digital Systems Using and Applicative Hardware Description Language", M.S. Thesis, UCLA Computer Science Department, Technical Report No. CSD-840046, November 1984.

[Mes85] F. Meshkinpour and M.D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs", IEEE Proc. of the 22nd ACM/IEEE Design Automation Conference, 1985, pp.238-244.

[Pat84] D. R. Patel, "Applicative Languages in the Specification and Implementation of VLSI-oriented Hardware Algorithms", Ph.D. Dissertation (in progress), UCLA Computer Science Department, 1984.

[Pat85] D.R. Patel, M. Schlag, and M.D. Ercegovac, "vFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms", Proc. 1985 ACM Conference on Functional Programming Languages and Architectures, Nancy, France, Springer-Verlag Lecture Notes 201, 1985.

[She84] M. Sheeran, "muFP, a language for VLSI design", Proc. 1984 ACM Conference on LISP and Functional Programming, pp.104-112, 1984.

[Sch84] M.D.F. Schlag, "Extracting Geometry from FP Expressions for VLSI Layout", UCLA Computer Science Report CSD-840043, October 1984.

[Sch86] M.D.F. Schlag, "Layout from a Topological Description", Ph.D. Dissertation, UCLA Computer Science Department, Technical Report CSD-860039, July 1986.

[Slu84] E. Slutz et al., "BDL: A Hierarchical Block Design Language", Proc. of the 21st IEEE Conference on Design Automation, pp.59-65, June 1984.

[Wor86] J. Worley, "A Functional Style Description of Digital Systems", M.S. Thesis, UCLA Computer Science Department, Technical Report CSD-860054, February 1986.

[Wu87] W. Wu, "Interface between UCLA FP System and VIVID", MS project in progress, UCLA Computer Science Department, 1987.

3. SUMMARY OF CONTRIBUTIONS: Task 2

The research in this task has focused on the application of low-cost arithmetic error codes [AVIZ 71],[AVIZ 81],[AVIZ 83] to the concurrent detection of errors (due to both transient and permanent faults) originating in systolic systems. A new generalization has been developed that extends the application of low-cost inverse residue codes into two dimensions: row (byte) and column (line) residues. [AVIZ 83]. This extension improves the detection of errors, especially of those due to indeterminate faults, and provides certain error-correction capabilities. Previous research investigated the advantages offered by two-dimensional inverse residue codes in the detection and correction of errors that affect byte-wide communication paths and systolic processing elements. Such paths are widely used in high-performance systolic arrays and for inter-processor communication in large multi-array systems.

In general, it has been shown that the remaining undetectable errors in the message X are those that are missed by *both* checks: modulo $2^B - 1$ over the bytes (not including the check line bits), and modulo $2^{k+1} - 1$ over the lines, with the check byte bits included in each line. Most unidirectional errors are detectable; furthermore, the detection of bidirectional errors is significantly improved. A single-line correcting, double-line detecting property was also demonstrated for unidirectional errors.

The research has led to the development of the fundamental byte-serial arithmetic algorithms for operands encoded in two-dimensional low-cost inverse residue codes. The algorithms are:

- (a) the line-residue checking algorithm;
- (b) the additive inverse (complementation) algorithm;
- (c) the addition algorithm.

The details of the algorithms have been presented in [AVIZ 85]. It has been shown that byte-serial arithmetic can be carried out with operands which are encoded in two-dimensional residue and inverse residue codes. Two-dimensional encodings provide a very powerful error-detecting and a substantial error-correcting capability for byte-serial arithmetic. Promising application areas are systolic arrays, multiple-precision arithmetic, and high-speed array computing.

References

- [AVIZ.71] Algirdas Avižienis "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers*, C-20: 1322-1331, November 1971.

[AVIZ 81] Algirdas Avižienis "Low-Cost Residue and Inverse Residue Error-detecting Codes for Signed-Digit Arithmetic," *Proceedings, 5th Symposium on Computer Arithmetic*, 1981, pp. 165-168.

[AVIZ 83] Algirdas Avižienis and C. S. Raghavendra, "Applications for Arithmetic Error Codes in Large, High-Performance Computers," *Proceedings, 6th IEEE Symposium on Computer Arithmetic*, June 1983, pp. 169-173.

[AVIZ 85] Algirdas Avižienis "Arithmetic Algorithms for Operands Encoded in Two-Dimensional Low-Cost Arithmetic Error Codes", *Proc. 7th IEEE Symposium on Computer Arithmetic*, May 1985, pp. 285-292.

PUBLICATIONS RESULTING FROM THIS PROJECT

L. Alkalaj, "A Uniprocessor Implementation of FP Functional Language", MS Thesis, UCLA Computer Science Department TR CSD-860064, April 1986.

Alkalaj, L., M.D. Ercegovac, and T. Lang, "A Dynamic Memory Management Policy for FP", 1987 Hawaii International Conference on Systems Science.

A. Avižienis "Arithmetic Algorithms for Operands Encoded in Two-Dimensional Low-Cost Arithmetic Error Codes", *Proc. 7th IEEE Symposium on Computer Arithmetic*, May 1985, pp. 285-292.

J. Arabe, "Compiler Considerations and Run-Time Storage Management for a Functional Programming System", PhD Dissertation, UCLA Computer Science Department, CSD TR 86004, August 1986.

M. Ercegovac and T. Lang, "A Division Algorithm with Prediction of Quotient Digits", the 7th IEEE Symposium on Computer Arithmetic, 1985.

F. Meshkinpour, "On Specification and Design of Digital Systems Using and Applicative Hardware Description Language", M.S. Thesis, UCLA Computer Science Department, Technical Report No. CSD-840046, November 1984.

F. Meshkinpour and M.D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs", IEEE Proc. of the 22nd ACM/IEEE Design Automation Conference, 1985, pp.238-244.

J. Moreno, "Analysis of Alternatives for a Singular Value Decomposition Problem", UCLA Computer Science Department, M.S. Thesis, TR CSD-850035, October 1985.

J. Moreno, "A Proposal for the Systematic Design of Arrays for Matrix Computations", UCLA Computer Science Department, TR CSD 870019, May 1987.

D. Patel, "A VLSI Design Environment Based on an Applicative Language", PhD Dissertation (Proposal), UCLA Computer Science Department, 1984.

D.R. Patel and M.D. Ercegovac, "A High-Level Language for Hardware-Oriented Algorithm Design", presented at the Symposium "Electronics and Communications in the 80's", Indian Institute of Technology, Bombay, February 1983.

D.R. Patel, M. Schlag, and M.D. Ercegovac, "vFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms", Proc. 1985 ACM Conference on Functional Programming Languages and Architectures, Nancy, France, Springer-Verlag Lecture Notes 201, 1985.

S. Pungsornruk, "A Threaded FP Interpreter/Compiler", M.S. Thesis, UCLA Computer Science Department TR CSD-860061, March 1986.

M. Sausville, "Gathering Performance Statistics on Hardware Specified in FP", UCLA Computer Science Department Internal Report, March 20, 1986.

M. Schlag, "Layout from a Topological Description", PhD Dissertation, UCLA Computer Science Department, Summer 1986.

M.D.F. Schlag, "Extracting Geometry from FP Expressions for VLSI Layout", UCLA Computer Science Report CSD-840043, October 1984.

J. Worley, "A Functional Style Description of Digital Systems", M.S. Thesis, UCLA Computer Science Department, Technical Report CSD-860054, February 1986.

**vFP: An Environment for the Multi-level
Specification, Analysis, and Synthesis
of Hardware Algorithms** [†]

Dorab Patel, Martine Schlag and Milos Ercegovac
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA

Abstract

This paper describes a method based on applicative languages for the specification, evaluation and synthesis of hardware algorithms. The goal of the research effort is to provide designers with an environment in which they can rapidly explore alternative designs for their algorithms throughout the synthesis process. It is possible to specify the algorithm at arbitrary levels of abstraction and have the system rapidly evaluate certain parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. Layouts which are suitable as floor plans are extracted from high-level algorithms.

1 Introduction

The complexity of VLSI design can only be managed by the application of CAD tools at all levels of the design process. In order to be effective, these tools must be flexible enough to be tailored to any specific design. Generally, VLSI CAD tools may be distinguished as being of either or both of two types: bottom-up composition tools or top-down synthesis tools. For bottom-up composition tools, the user either exactly specifies the placement of modules and the interconnections between them, or relinquishes control over the layout to the tool's algorithm. Examples of composition tools are graphic layout editors (e.g. Caesar, Magic) [Ousterhout81, Ousterhout84] and placement and routing tools [Rivest82]. Top-down synthesis tools are capable of generating layouts from high-level specifications. Examples would include various register-transfer silicon compilers that have been proposed and built [Sistkind82, Director81, Johannsen79]. Generally, these tools do not provide any estimate of the area or delays of the circuit during the synthesis process. That is, designers do not know the effects of their decisions on the performance until the design is complete.

Many of the current design approaches were largely developed for SSI/MSI technologies and are limited because of:

- Lack of paradigms to deal with topological and geometrical aspects of algorithm design in a hierarchical, multi-level fashion.
- Lack of adequate methods to deal with communication requirements of VLSI implementations during a multi-level algorithm design process.
- Lack of an adequate interface to lower-level VLSI CAD tools; most systems require a logic

[†] Appeared in the Proceedings of the 1985 Functional Programming Languages and Computer Architecture Conference, Nancy, France, J.P. Jouannaud (Ed.) Lecture Notes in Computer Science 201, Springer-Verlag, 1985, pp. 238-255.

diagram at the entry level, thus forcing designers to cope with details which are apt to be changed later in the implementation process.

- Lack of visual feedback: graphical representations, generated automatically from a high-level algorithm, showing details selected by the designer are highly desirable.

This paper describes a method based on applicative languages [Backus78] for the specification, evaluation and synthesis of hardware algorithms. This method is supported by a set of tools that is being developed at UCLA. The goal of this effort is to provide designers with an environment in which they can rapidly explore various alternative designs for their algorithms. Thus, it is possible to specify the algorithm at any arbitrary level of abstraction and have the system rapidly evaluate performance parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. The advantage of using an applicative language is that it ties together the specification of the algorithm, the synthesis of the circuit and the evaluation of the implementation.

Others have explored incorporating applicative languages in VLSI design and have shown them to be viable. Lahti [Lahti81] used an applicative language to describe various combinational hardware structures. Johnson [Johnson84] utilized a demand-driven applicative language to describe and synthesize sequential digital circuits. Cardelli and Plotkin [Cardelli81] take a formal approach to describing sequential circuits with an emphasis on verification. Meshkinpour [Meshkinpour85] and Sheeran [Sheeran84] extended Backus' FP language with operators to handle sequential circuits.

2 Brief Introduction to vFP

vFP extends the language FP proposed by Backus [Backus78] with additional functional forms and primitives. In contrast to μ FP [Sheeran84], which extends FP's semantics to operate on streams, the semantics of vFP are the same as those of FP when it is used to specify algorithms. A program in vFP (as in FP) is a function that maps objects into objects. Objects are either atomic (numbers or strings) or sequences of objects. The distinguished atom denotes an undefined value. By definition, any sequence which contains as an element is itself undefined and thus equal to . The primitive functions of vFP consist of

arithmetic functions,	$+ : (1,5) \rightarrow 6$	$\cdot : (3,2) \rightarrow 6$
logical functions,	$\text{andg} : (1,0) \rightarrow 0$	$\text{org} : (0,0) \rightarrow 0$
predicates,	$\text{atom} : (1,2) \rightarrow F$	$= : (3,3) \rightarrow T$
selector functions,	$3 : (2,(4,5),6,(8,(9,10))) \rightarrow 6$	$\text{last} : (1,4,6) \rightarrow 6$

and structure modifying functions.

$\text{trans} : ((1,2,3),(4,5,6)) \rightarrow ((1,4),(2,5),(3,6))$	$\text{apndl} : ((1,2,3,4)) \rightarrow (1,2,3,4)$
$\text{distl} : (x, (a,b,c)) \rightarrow ((x,a),(x,b),(x,c))$	$\text{distr} : ((a,b,c),x) \rightarrow ((a,x),(b,x),(c,x))$

Functional forms are used to combine primitive functions into more complex functions.

compose	$(f @ g) : x \rightarrow f : (g : x)$
construct	$[f,g,h] : x \rightarrow (f:x, g:x, h:x)$
apply to all	$\&f : (p,q,r) \rightarrow (f:p, f:q, f:r)$
constant	$\%k : x \rightarrow k \text{ if } x \text{ is not}$
right insert	$'f : (x_1, \dots, x_n) \rightarrow f : (x_1, 'f : (x_2, \dots, x_n))$
tree insert	$f : (x_1, \dots, x_n) \rightarrow f : (f : (x_1, \dots, x_{n/2}), f : (x_{n/2+1}, \dots, x_n))$

A major syntactic difference between vFP and Backus' FP is that parameters to functions may be named and then referred to in the function body with the same restrictions as described in [Backus81]. In addition, the arithmetic, logical, and predicate functions may be used either in a prefix or an infix manner. This improves the readability of hardware specifications. For example, the following definition of a *FullAdder* in Backus' FP.

```
FullAdder =  
  (org@[org@[andg@[1.2],andg@[2.3]],andg@[1.3]], xorg@[1.xorg@[2.3]])  
could alternatively be written in vFP as  
defun FullAdder(a,b,Cin)  
  (((a andg b) org (b andg Cin)) org (a andg Cin), a xorg (b xorg Cin))  
enddef
```

Owing to the natural specification of parallelism in FP-like languages, they are suited to describing parallel hardware algorithms. These specifications are executable. Since such programs are *referentially transparent*, it is possible to have an algebra of programs which may be used to reason about their behavior. These methods may be used in conjunction with each other to convince the designer that the program implements the envisioned algorithm. Specifications can also be executed symbolically using a symbolic input during which it is possible to extract the topological structure of the algorithm. Therefore, there is a direct relationship between the structure of an algorithm written in vFP and the planar topology of its layout.

3 Algorithm Synthesis in vFP

The designer first specifies the algorithm in vFP. The algebra of vFP programs may be used to reason about the algorithm. In addition, the specification may be executed with sample data to validate the program.

vFP can be used to describe circuits at various levels of abstraction. Designers are free to choose whichever level is "best" for their current purposes. At some higher level of abstraction, the structure of the *FullAdder* may not be relevant, and thus the definitions given earlier would suffice to describe its behavior. At a lower level of abstraction, where the structure of a function is to be considered, an alternative definition which has a different structure may be substituted. For example, the *FullAdder* could be defined in terms of *HalfAdders*.

```
defun FullAdder(a,b,Cin)  
  {1 or 2, 3} @ apndl @ [1, HalfAdder@[2.3]] @ apndr @ [HalfAdder@[a,b], Cin]  
enddef  
  
defun HalfAdder(a,b) [a andg b, a xorg b] enddef
```

Transformations may be used to refine the program to whatever level of detail is required. In this way it is possible to first specify the algorithm at a level of abstraction that is high enough to aid validation and then refine it to the level at which it can be easily implemented.

4 The Evaluation of vFP Algorithms

It is possible to tag selected user-defined functions so that when a vFP specification is executed an estimate of the performance of the algorithm can be provided. Tagging a function tells the system that this is a function of interest at the current level of abstraction. As the execution proceeds, the interpreter keeps track of the level at which a tagged function is executed.

The level of a tagged function is defined as one plus the maximum of the levels associated with the atoms in its input object. The level of each atom is initially zero. Each time a tagged function is encountered, its level is determined and is assigned to the atoms in its output object. However, there is a problem when a tagged function occurs within another tagged function. In this case the level of the inner function is determined with respect to the outer function resulting in a hierarchy of levels. This is accomplished by assigning the level zero to each atom of the outer function's input object, and computing the level of tagged functions as before until the computation of the outer function has been completed. The level of the outer function and the atoms in its output object is determined as before and hence is independent of whether or not any tagged function occurs within the outer function. Levels are used to predict the speed at which the circuit would perform, to obtain an idea of where the parallelism in the algorithm is, and to get an estimate of the area that would be occupied by the circuit. A better estimate of the area is obtained by methods mentioned in the next section.

This capability of having the system estimate performance parameters is useful in tradeoff analyses. For example, consider the following function:

$$z = \begin{cases} 1 & \text{if } a = (b-1) \bmod 8 \\ 2 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

schemeA and schemeB, below, are two algorithms for implementing the function. If the boolean functions (andg, org, norg, and xorg) are tagged, the results shown in Figure 1 are obtained.

```
# scheme A  inputs : ((a) (b))  outputs : (z1 z0)
defun schemeA
  &(org@&andg@trans)@distl@[1,[id,rot]@2]@&decoder
enddef

# scheme B  inputs : ((a) (b))  outputs : (z1 z0)
defun schemeB(a,b)
  &compare @ distl @ [a, [b, d @ adder @ [b, [%1, %1, %1]]]]
enddef

defun compare  norg@org@&xorg@trans  enddef

defun adder
  apndl@[xorg@(xorg@1,1@2),d@2]
  @[(1,!add@apndr@[lrl_HalfAdder@last]@d) @ trans
enddef

defun add(a,b)  concat@[FullAdder@apndr@[a,1@b],d@b]  enddef
```

# statistics for schemeA					# statistics for schemeB				
level	Andg	Org	Notg		level	Andg	Xorg	Org	Notg
1	2		6		1	2	6		
2	10				2	1	2	1	
3	14				3		1	2	
4	14				4		1		1
5		8			5		1		
6		4			6			1	
7		2			7			1	
Totals	40	14	6		8				1
					Totals	3	11	5	2

Figure 1: A comparison of two implementations

The results in Figure 1 show that *schemeA* uses a total of 60 gates, while *schemeB* uses a total of 21 gates. However, it is to be noted that 11 of the 21 gates are xor gates which would normally occupy a larger area. Given an estimate of the area occupied for each of the gates, it is possible to have an estimate of the area occupied by each implementation. Since *schemeA* has 7 levels while *schemeB* has 8, *schemeA* would be faster than *schemeB* under the assumption that all the tagged functions had the same delay.

In addition to the time and space estimates provided by the *level* mechanism, the system can be extended to allow the specification and calculation of user-specified parameters for each tagged function and for the algorithm as a whole.

5 Space Domain Implementations of vFP Algorithms

A vFP algorithm can be mapped into a structure corresponding to a combinational network by passing symbolic inputs to functions which in turn generate symbolic outputs. The unit of information represented by a symbolic atom can be anything corresponding to the level of abstraction. Thus, a symbolic atom may represent a wire, a set of wires, a bit vector, or an integer, as required. An acyclic computation graph with vFP primitives as nodes is obtained by tracing the application of a function to a symbolic input. This computation graph can be transformed into a layout using techniques described later. By tagging the appropriate functions, the layout may be generated at any desired hierarchical level. For example, Figure 2 shows a layout of a *FullAdder* using *and*, *or*, and *xor* gates; whereas Figure 3 shows the same *FullAdder* as being composed of *HalfAdders*.

The mapping from a vFP algorithm to a combinational network is allowed under the following restrictions. Functions like *iora*, whose output structure depends on an input value, cannot be laid out. In addition, during symbolic evaluation, the predicate part of a conditional must be evaluable to a boolean.

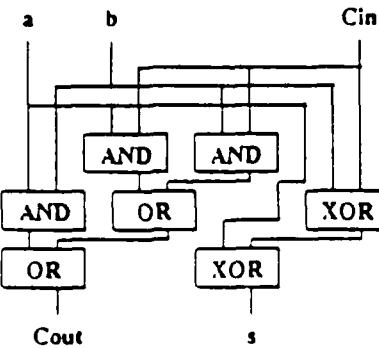


Figure 2: The structure of a FullAdder

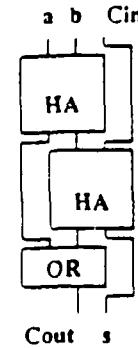


Figure 3: A FullAdder using HalfAdders

As in μ FP (Sheeran84), *structural iterations* over the input of the circuit can be handled by the *insert* and *apply-to-all* functional forms. Other types of *structural recursions* are allowed in vFP since the *conditional* functional form is treated as a *structural* form for the purposes of layout. Depending on the value of the predicate of the conditional, either the consequent or the alternate part will be evaluated symbolically for its structure but no structure will be generated for the predicate part. A new primitive called *sw* (for switch) is provided in vFP which corresponds to the conditional form in μ FP. This primitive takes three arguments. If the first is 1 then the output is the second argument; if it is 0 then the output is the third argument; else it is . In addition, it is required that the *structures* of the second and third arguments be the same.

A vFP description of a circuit can be generic in the sense that the description is independent of the input dimensions of the circuit. For example, there needs to be only one description of a decoder. This same description works for a decoder independent of the number of inputs. The 3-to-8 decoder shown in Figure 4 is obtained by evaluating the description of the generic decoder with a symbolic argument of size 3. Figure 4 shows how the generic iterative decoder is formed by first applying 1-to-2 decoders (*Dec1*) to the inputs and then inserting the function *DecStage*. *DecStage* takes an n-to- 2^n decoder and a new input to make a $(n+1)$ -to- 2^{n+1} decoder.

```
defun Decoder !DecStage @ &Dec1 enddef
```

```
defun DecStage &anv'g @ concat @ &distl @ distr enddef
```

```
defun Dec1 [not{id}] enddef
```

As before, these implementations may be evaluated to get speed/area estimates, but now, since routing is taken into account, a better estimate of area can be provided.

Cell iterative networks are combinational circuits which are formed by interconnecting a particular cell in a regular pattern. Although combinational circuits without feedback can be described in vFP using the forms inherited from FP, some additional functional forms are provided to give designers more control over exactly how cell iterative networks are to be laid out. These networks are thus

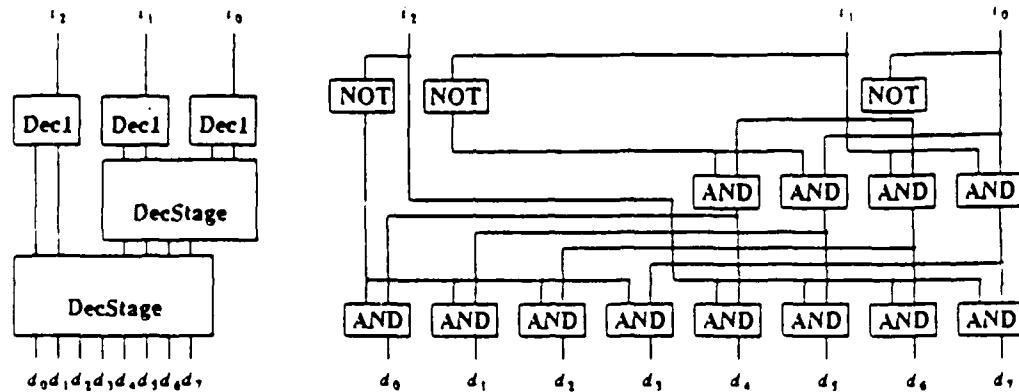


Figure 4: A 3-to-8 Decoder at different levels of abstraction

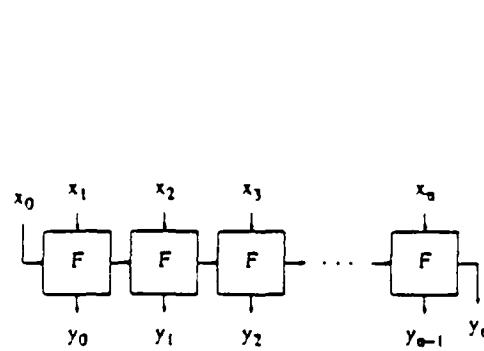


Figure 5: The seq functional form applied to F

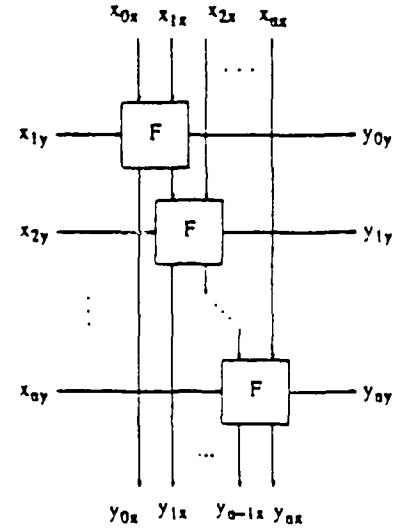


Figure 6: The seqxy functional form applied to F

readily described in vFP by invoking the form (corresponding to the interconnection pattern) on the function (corresponding to the cell). For example, Figure 5 shows the seq functional form pictorially. Sometimes it is necessary to have two inputs to the function F at each stage and to have one of those inputs come in from the x direction and the other from the y direction. This is accomplished by the seqxy functional form shown in Figure 6. Though both forms result in the same computation graph, their layout is different.

system in vFP. D^{-1} is a phantom element that corresponds to an inverse time delay. It is used to keep track of the number of clock pulses the output is going to be delayed by. This information is needed by the *construct* functional form to synchronize its components since the semantics of the construct require that the outputs of its elements appear together. Generally the D^{-1} elements are moved, via transformations, to the outputs of the circuit where they serve to denote the delay.

When elements of a sequence are available serially in time along the same wire(s), it is necessary to know when each element is valid. This is accomplished, during symbolic simulation, by having each symbolic item carry the name of a clock with it. It is assumed that its value will be stable before every tick of the named clock. The system will automatically widen the intervals between clock ticks to ensure that this is true. Initially, all the inputs are associated with the same clock. Each combinational element will assign to its output the clock associated with its input. If there are n elements to the input sequence of a *SOP1*, then each of its output elements will be clocked by the clock nC_k ; and conversely for a *POSI*. A clock named nC_k denotes a clock which has n clock ticks in between consecutive ticks of the clock named C_k . Though the description of a *SOP1* or *POSI* is generic, the value of n (the number of elements in the sequence) must be known at layout time.

As an example, consider a time-domain implementation of an inner-product algorithm ($!+ @ \& *$). The straightforward implementation of the algorithm, using the equations given above, would result in the layout shown in Figure 7. Since there are two D^{-1} elements in the layout, the output will be delayed by two clock ticks from the input.

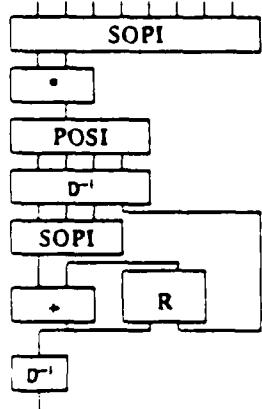


Figure 7: Initial Implementation of the Inner Product

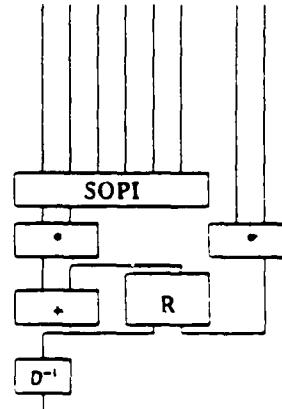


Figure 8: Optimized Implementation of the Inner Product

However, using the identities

$$D^{-1} @ POSI @ f @ SOPI = apndr @ [D^{-1} @ POSI @ \& f @ SOPI @ dr, f @ last]$$

$POSI @ D^{-1} @ SOPI = SOPI @ D^{-1} @ POSI = id$

$f @ D^{-1} = D^{-1} @ f$

$(tr, last) @ apndr = apndr @ (tr, last) = id$

the program may be transformed into the following

$D^{-1} @ !^+ @ apndr @ (&^+ @ SOPI @ tr, * @ last)$

whose layout is shown in Figure 8. The single D^{-1} element denotes that the output is delayed only one clock tick from the input.

This implementation accepts all its inputs simultaneously and eventually gives its result. It will only work for input sequences of one particular length, since only fixed size *SOP*'s can be laid out. However, if each element of the input sequence was input serially to the implementation, a corresponding *POSI* could be introduced at the input and then used to transform out the *SOP* that exists in the current implementation. This would make the implementation generic in the sense that it would be able to handle arbitrary length sequences as its inputs.

7 Layouts from vFP Specifications

In this section the mapping from vFP algorithms to layouts is briefly described. A more detailed exposition and a description of its implementation can be found in [Schlag84]. The intent of this system is to provide the vFP designer with an interactive environment in which the design can be viewed as it is constructed. The mapping from vFP is actually the composition of two mappings. An vFP function is first mapped to an intermediate form (IF) which reflects the planar topology of the function and then this IF is mapped to fixed geometry by selecting and resolving relative position constraints (compaction).

The rationale for dividing the mapping in two steps is the observation that a certain portion of the mapping from vFP should be functional even though the entire map cannot be. That is, a particular vFP function applied to a particular symbolic object should define an IF uniquely, while the geometry of the function should depend on its environment. Fixing the geometry of a sub-function may create wiring and shape incompatibilities with other sub-functions which would require additional area to resolve. Functionality has two advantages.

1. The mapping can be implemented as an application of an vFP function to an object.
2. Algebraic transformations on the vFP function have predictable effects on the IF.

The extraction of the topology (IF) of an vFP expression is implemented as an interpreter. A function applied to an object generates an IF and each combining form dictates a topological organization of the IFs of its sub-functions. The routing is the direct result of the routing primitives and the combining forms of the function. This implementation generates a sketch of the vFP specification in terms of "boxes" and "wires" by symbolically tracing the vFP function and representing each atom as a wire. The level of abstraction of the sketch can be controlled by selecting which functions to represent as

boxes and what objects each atom represents. The IF consists of a list of horizontal cross-sections each of which is a left to right ordering of the "boxes" and "wires" which intersect the cross-section. Any cross-overs are represented explicitly: each cross-section corresponds to a horizontal track. The IF generated by the interpreter is fed to a program which resolves the cross-sections using horizontal compaction and displays the sketch on a graphics terminal.

An example is presented to illustrate how vFP facilitates the transition from algorithm to implementation taking into account the layout. The vFP specification of a carry chain adder [Brent80] is considered. The specification is generic in that it adds two bit vectors of length 2^n for $n > 0$. The input consists of the 2^n pairs of bits to be added with the leftmost pair, containing the least significant bits, $((a_1, b_1), (a_2, b_2), (a_3, b_3), \dots, (a_{2^n}, b_{2^n}))$.

For $1 \leq i, j \leq 2^n$,

$$P_{i,j} = \begin{cases} 1 & \text{if a carry into column } i \text{ would} \\ & \text{propagate as a carry out of column } j \text{ and } G_{i,j} = \begin{cases} 1 & \text{if adding columns } i \text{ through } j \\ & \text{causes a carry out of column } j \\ 0 & \text{otherwise} \end{cases} \\ 0 & \text{otherwise} \end{cases}$$

The computation is performed by computing the carries for each column, $G_{i,j}$, and then obtaining the sum bit using,

$$s_i = G_{1,i-1}P_{i,i} \text{ for } 1 < i \leq 2^n, \quad s_1 = P_{1,1}, \quad \text{and } s_{2^n+1} = G_{1,2^n} \quad [7.1]$$

$P_{i,j}$ and $G_{i,j}$ are computed for each i by using the following identities, implemented by the function PG.

$$\text{For } i \leq j < h, \quad P_{i,h} = P_{i,j}P_{j+1,h} \quad \text{and} \quad P_{i,h} = (G_{i,j}P_{j+1,h}) + G_{j+1,h} \quad [7.2]$$

The initial $P_{i,j}$ and $G_{i,j}$ are computed by the function PG1.

$$P_{i,j} = a_i b_i, \quad G_{i,j} = a_i b_i \quad [7.3]$$

The computation of $P_{i,j}$ and $G_{i,j}$ is achieved in two steps by the function getcarries. The following is the specification of getcarries.

```

# input = ((a0,b0),(a1,b1),(a2,b2),..., (a2**n - 1,b2**n - 1))
defun getcarries  secondhalf@split@1@firsthalf@&PG1  enddef

defun firsthalf  if eq@[length,%1] then id else firsthalf@&stage1@pair fi  enddef

defun stage1  concat@(&D@1,&D@tlr@2,[PG@[last@1,last@2]])  enddef

defun secondhalf  if eq@[length,%1]@1 then done else secondhalf@concat@
  [split@&D@1.stage2@tlr@apndr@[concat@&[id, last]@tlr, last],fi
enddef

defun stage2  concat@
  &([apndr@(&D@tlr@1@2.PG@[1.last@1@2]),&D@2@2]@[1.split@2])@pair
enddef

defun D id enddef

```

```
defun done id enddef
```

First *getcarries* computes $(P_{1,i}, G_{1,i})$ by applying PG1 to the two bits in each column and then it applies *firsthalf*. *firsthalf* computes $(P_{i-2^k+1,i}, G_{i-2^k+1,i})$ for each column $i = (2m+1)2^k$ where m is an integer. This is accomplished by arranging each column (i.e. its pair (P, G)) in a group of its own and then recursively applying the function *stage1* to pairs of groups until only a single group remains. *stage1* combines a pair of groups computing a new (P, G) for the last column of the second group by applying PG to the last columns of the two groups; the pair of groups is then concatenated to form one group. All other columns are unchanged; the function D which is given the definition id is applied to them. When all columns are in a single group *getcarries* applies the function *secondhalf* to compute the final (P, G) 's. *secondhalf* is also recursive, terminating when each column is in a group by itself. At each step the final (P, G) 's of decreasing multiples of powers of 2 are computed. Assume that in the previous step $P_{1,i}$ and $G_{1,i}$ have been computed for each column $i = m2^k$. In the next step to compute the (P, G) 's of columns which are multiples of 2^{k-1} , it is necessary only to compute new (P, G) 's for columns $i = (2m+1)2^{k-1} = m2^k + 2^{k-1}$, the odd multiples of 2^{k-1} . The current (P, G) in column i is $(P_{i-2^{k-1},i}, G_{i-2^{k-1},i})$. $(P_{1,i}, G_{1,i})$ can be obtained by applying PG to the current (P, G) and $(P_{1,m2^k}, G_{1,m2^k})$. Initially the columns are divided into two groups and since *firsthalf* computed the final (P, G) 's for powers of 2, the last column (a multiple of 2^{k-1}) has its final value. At each step *secondhalf* duplicates the last column from each group and then applies *stage2* after removing the first group. *stage2* takes each group, splits it into two and computes new (P, G) 's for the last column in the left group of each new pair using the duplicated column immediately to the left of the group. The first group is then appended to the result of *stage2*.

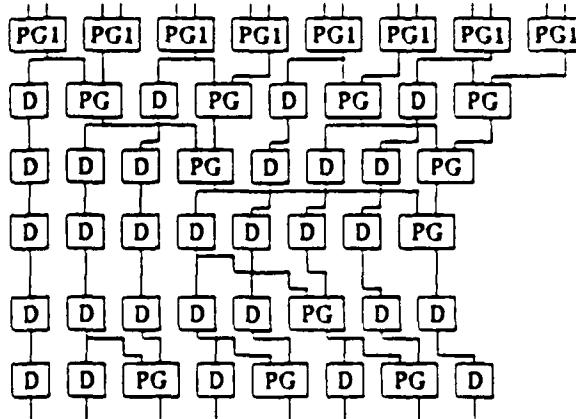


Figure 9: The sketch of *getcarries* with each (P, G) as a wire

Figure 9 is the sketch of *getcarries* in which the pair (P, G) for each column is represented by a single wire. This is accomplished by directing the interpreter to draw PG1, PG and D as boxes and by giving PG1 and PG symbolic definitions.

```
(define-symbolic PG1 input=(a b) output = (c))  
(define-symbolic PG input=(a b) output = c)
```

```

(drawbox PG1 label=PG1 ht=2)
(drawbox PG label=PG ht=2)
(drawbox D label=D ht=2)

```

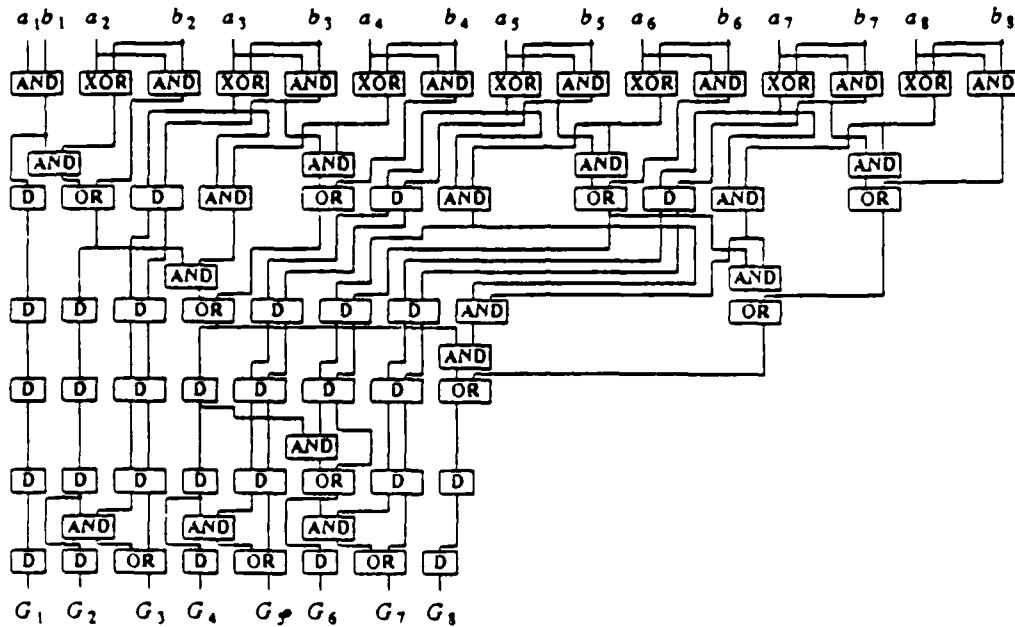


Figure 10: The sketch of getcarries with the definitions of PG1 and PG filled in

Figure 10 is the sketch of getcarries with each wire corresponding to a bit this time and with the specifications of the functions PG and PG1 "filled in." The definition of done is changed since only the G of each column is required to compute the final sum. Notice that the layout interpreter generates only those wires and boxes which have paths to an output. The previous specification would be extended as follows.

```

defun done &(2@1) enddef
defun PG1 [[xorg, andg]] enddef
defun PG [andg@[1@1,1@2], org@[andg@[2@1,1@2], 2@2]] enddef
(drawbox D label=D ht=2)

```

To obtain the final sum by (7.1) it is necessary to combine the first $P_{i,j}$ in each column, $P_{1,j}$, with the G of its left neighbor $G_{1,j-1}$. One way of doing this would be to duplicate each $P_{1,j}$ generated by PG1, route them along the side and then merge them back into the columns to compute the final sum as in Figure 11. The additional area required for routing makes this an unattractive alternative. A better design would be to route the $P_{1,j}$ along with the (P, G) down its own column. This extension is easily handled in vFP by modifying the function PG so that it duplicates its first argument if it receives only two arguments in a column and simply passes on the extra argument otherwise. The specification is modified as follows.

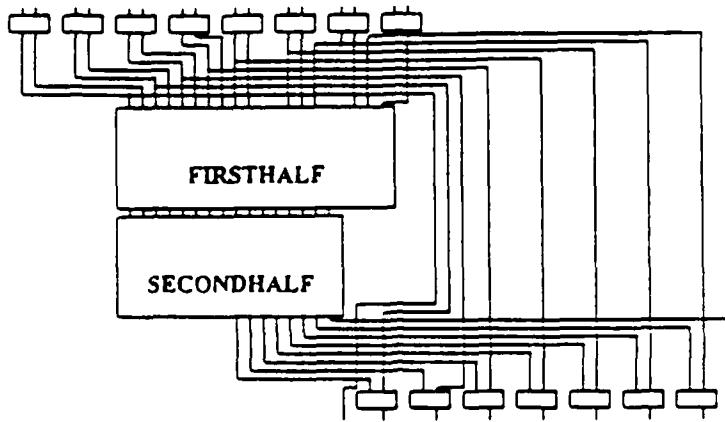


Figure 11: An inefficient design

```

defun add
  concat@[1].&xorg@pair@t@t@r,[last]
    @concat@apndl@[1@1.&([1,3]@1)@t]@getcarries
  enddef

defun PG
  apndl@[D@1@2, if null@t@t@2 then oldPG else oldPG@[1,t@2] fi]
    @ if null@t@t@1 then id else [t@1,2] fi
  enddef

defun oldPG  [andg@[1@1,1@2],org@[andg@[2@1,1@2],2@2]]  enddef

defun done id enddef

(drawbox D label=D ht=2)

```

The function `add` applies `getcarries` and then handles the columns according to (7.1) to obtain the final sum bits. Figure 12 is the sketch of `add`.

Figure 13 is the sketch obtained of a carry save array multiplier. This example is presented to illustrate the geometric flexibility of fixing only the planar topology in the specification. The functions `HA*`, `FA*`, `FA**`, and `HalfAdder` are represented as primitives. The specifications of the functions `HA*`, `FA*`, and `FA**` are

```

* FA* op2 : ((a b) (y x)) --> ((c x) s) where 2c + s = (a + b + yx)
defun op2    [[org@[1,1@2],3],2@2]@
  ([1@1,HalfAdder@[2@1,2],3]@[HalfAdder@[1@1, andg@2,2@2]
enddef

* HA* op1 : ((a) (y x)) --> ((c x) s) where 2c + s = (a + yx)
defun op1  [[1@1,2@1]@[HalfAdder@[1@1, andg@2,2@2]] enddef

```

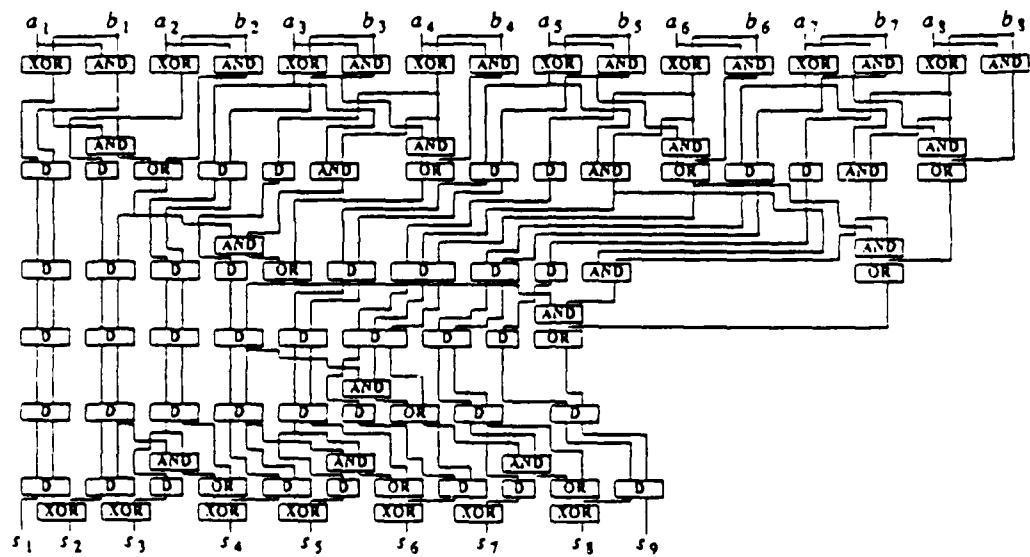


Figure 12: The sketch of add

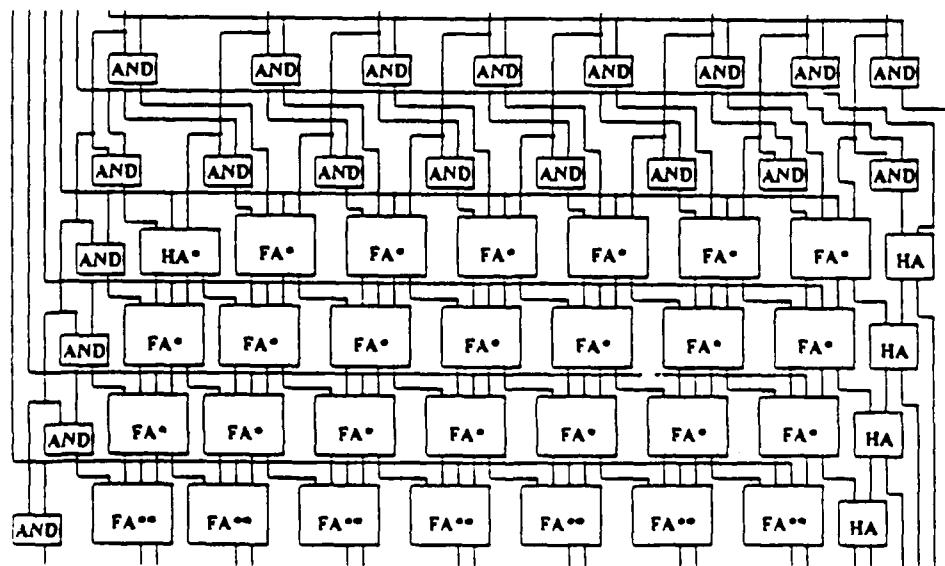


Figure 13: A Carry Save Array Multiplier with FA^* , HA^* and FA^{**} as primitives

```

* FA** lop2 : ((a b) (y x)) --> (c s) where 2c + s = (a + b + yx)
defun lop2
  [org@[1,1@2].2@2]@[1@1.HalfAdder@[2@1.2]]@[Half Adder@1.andg@2]
enddef

defun op0 [1.andg] enddef

```

Figure 14 contains the sketches of these functions, while in Figure 15 the same carry save array multiplier is represented in terms of lower level primitives. Note that the geometry of the functions HA*, FA* and FA** varies; each instance has some flexibility in adapting to the particular geometric constraints it encounters.

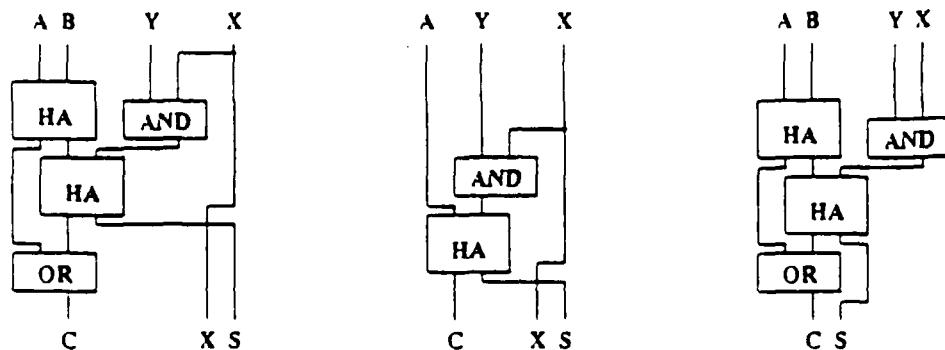


Figure 14: FA*, HA* and FA**

All of the figures in the paper with the exception of Figures 5 and 6 were generated by this system. It is limited in that the data flow is vertical with a function's inputs and outputs on the top or bottom. More sophisticated layout techniques which would not suffer from this limitation are being examined. However this system is useful in that it provides visual feedback quickly allowing the designer to see the planar implications of the specification.

8 Concluding Remarks

The objective of this research is to develop a formal high-level language approach to specification, simulation, performance evaluation, and chip layout planning for VLSI systems. Our approach takes a high-level applicative language (vFP) and programming style as its basis. The rationales for using vFP and its potential in dealing with several specification and implementation aspects are the subject of this paper. Specifically, a few examples have illustrated how vFP can be used to specify combinational, iterative, and sequential circuits. User-specifiable performance parameters may be used at any abstraction level to provide a basis for making design decisions during the synthesis process. Layouts which are suitable as floor plans are extracted from high-level algorithms. Currently, an automated attribute system is under development. More sophisticated layout techniques and topological optimizations are being examined, as are techniques to handle other classes of sequential circuits.

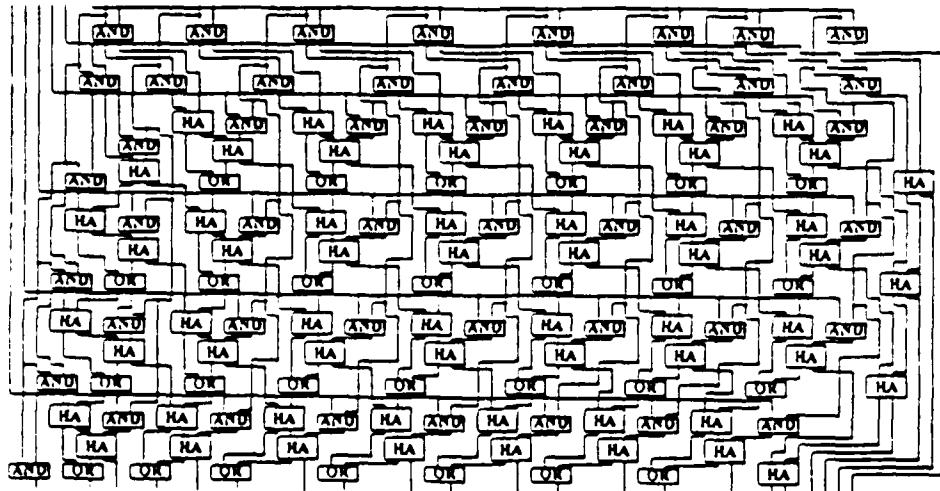


Figure 15: The Carry Save Array Multiplier with FA*, HA* and FA** filled in

9 Acknowledgements

This work was supported in part by ONR Contract N00014-83-K-0493, and by Rockwell/UC MICRO Grant 157. The presentation and content of this paper has benefited greatly from the detailed comments provided by one of the referees.

REFERENCES

- [Backus78] John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM* 21(8), pp. 613-641 (August 1978) 1977 ACM Turing award lecture.
- [Backus81] John Backus, "The Algebra of Functional Programs: Function level Reasoning, Linear Equations, and Extended Definitions," *Proceedings International Colloquium on Formalization of Programming Concepts Lecture Notes in Computer Science #107*, pp. 1-43, Springer Verlag (1981).
- [Brent80] R. P. Brent and H. T. Kung, "The Chip Complexity of Binary Arithmetic," *Proceedings 12th ACM Symposium on the Theory of Computing*, pp. 190-200 (May 1980).

[Cardelli81] Luca Cardelli and Gordon Plotkin, "An Algebraic Approach to VLSI Design," pp. 173-192 in *VLSI 81 - Very Large Scale Integration First International Conference on VLSI*, ed. John P. Gray (1981).

[Director81] S. Director, A. Parker, D. Siewiorek, and D. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Transactions Circuits and Systems CAS-28(7)*, pp. 634-645 (July 1981).

[Johannsen79] David Johannsen, "Bristle Blocks: A Silicon Compiler," *Proceedings 16th Design Automation Conference*, pp. 310-313 (June 1979).

[Johnson84] Steven Johnson, *Synthesis of Digital Designs from Recursion Equations*, MIT Press (1984).

[Lahd81] D. O. Lahd, "Applications of a Functional Programming Language," Tech. Rep. CSD-810403, UCLA Computer Science Department, Los Angeles, California, (April 1981).

[Meshkinpour85] F. Meshkinpour and M. D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs," *Proceedings of the 22nd Design Automation Conference*, pp. 238-244 (June 23-26, 1985).

[Ousterhout81] John Ousterhout, "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design* II(4), pp. 34-38 (fourth quarter 1981).

[Ousterhout84] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI Layout System," *Proceedings of the 21st Design Automation Conference*, pp. 152-159 (June 25-27, 1984).

[Rivest82] Ronald L. Rivest, "The 'PI' (Placement and Interconnect) System," *Proceedings of the 19th Design Automation Conference*, pp. 475-481 (June 1982).

[Schlag84] Martine Schlag, "Extracting Geometry from FP for VLSI Layout," Tech. Rep. CSD-840043, UCLA Computer Science Department, Los Angeles, California, (October 1984).

[Sheeran84] Mary Sheeran, "muFP, a language for VLSI design," *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pp. 104-112 (August 6-8, 1984).

[Siskind82] Jeffrey Mark Siskind, Jay Roger Southard, and Kenneth Walter Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," *1982 MIT Conference on Advanced Research in VLSI*, pp. 28-40 (January 1982).

**A FUNCTIONAL LANGUAGE FOR DESCRIPTION
AND DESIGN OF DIGITAL SYSTEMS:
SEQUENTIAL CONSTRUCTS**

**F. Meshkinpour
M.D. Ercegovac**

**Reprinted from IEEE PROCEEDINGS OF THE 22ND
ACM/IEEE DESIGN AUTOMATION CONFERENCE,
Las Vegas, Nevada, June 23-26, 1985**

A Functional Language for Description and Design of Digital Systems: Sequential Constructs

F. Meshkinpour* and M.D. Ercegovac

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024, USA

Abstract

A functional (applicative) hardware description language (FHD), capable of dealing with both the sequential and combinational systems is discussed. The language supports multi-level executable specifications and interpretation of functional specifications as implementations at a given level of primitives. That is, the FHD specifications are symbolically interpreted to produce structural representations (implementations) of hardware algorithms. The symbolic interpreter presently implements the specification of hardware algorithms at the gate level. The FHD allows definition of function attributes, such as delay and number of logic level so that the performance characteristics of implementations can be obtained during simulation.

1. Introduction

Background: High-level hardware description languages (HDL) are used in various phases of design in order to reduce the design time and errors, and simplify checking, debugging and modification of specifications and the corresponding implementations [19]. The high-level HDLs are also used in simulation at various levels in the design hierarchy. Multi-level simulation is a very important aspect of VLSI design because of the lengthy manufacturing process.

The HDLs have been following the evolution of programming languages in the sense that both the imperative (procedural) and applicative (functional, non-procedural) languages have been considered as models. The HDLs based on conventional, imperative languages have several serious deficiencies: they have no rigorous basis, their syntax and semantics are complicated, their constructs are ad hoc, and they reflect closely the sequential model of computation. Consequently, the HDL programs tend to be complex and error-prone, difficult to compose out of other programs, provide no inherent basis for checking and verification, and do not support concurrency. Moreover, there is no direct correlation between a high-level specification and its imple-

* Currently with Perceptronics Inc., 6271 Variel Ave., Woodland Hills, CA 91367.

mentation at a topological/geometrical level. Use of variables and the possibility of side-effects make the analysis of algorithms and their implementations very difficult. The chief advantage of conventional HDLs is a common familiarity and wide-spread use of conventional languages.

Recognizing the potential benefits of languages with formal foundations, simple and precise semantics, and inherent power to deal with concurrency and multi-level abstractions, several researchers have considered nonconventional language approaches for specification and design of digital systems. Functional (applicative) programming languages [1,2] satisfy these properties [13,6,17,11]. The basis of functional programming (FP) style is the representation of computations by functions that map objects into objects, and functional forms that combine functions. Objects are atoms (e.g., numbers and strings) or sequences of objects. Since an FP program is a function, it provides a generic specification of a computation: it applies to any size of the input object. An FP language allows hierarchical description of computations. Since the language does not have side-effects, the composition and analysis of programs are straightforward. [1,6,17] A possibly the most significant property is the mathematical basis of FP languages which provides means for systematic program transformations and formal design verification.

The FP hardware description languages provide an integrated framework for the following phases of design: (i) Specification: capturing proper behavior, (ii) Implementation: obtaining a suitable structure (implementation), and (iii) Optimization: refinement of the implementation to satisfy realization constraints.

The differences between functional programming languages (FP) and imperative languages as general programming languages have been discussed in depth in [1,3]. The previous work on the use of functional languages as HDLs by [13,9,8,4,10,11,16,15] discusses key ideas and the tradeoffs of this class of programming languages compared to imperative HDLs. A functional program, being an expression, is clearly suitable for describing combinational networks. Most of the previous work on functional languages as HDLs dealt with the combinational systems only. Recently ap-

proaches to deal with sequential networks have been discussed in [15, 20].

Overview of the article: The principal contributions discussed here are: (i) FP language extensions to deal with sequential networks, and (ii) an attribute system to deal in a general way with the evaluation of characteristics of designs. Section 2 discusses the FP language (FHD), its sequential constructs, and the handling of attributes. The main features that support transformations from the algorithmic level into the logic design level are emphasized. Several functional forms have been added to support the specification of sequential systems. We illustrate these forms in both the behavioral and structural domains.

To aid the designer in estimating the performance parameters at various levels of abstraction of the designs obtained from FHD specifications, FHD provides means of defining and evaluating the system characteristics using attributes such as propagation delay and the number of logic levels. The paper concludes with a more complex example in Section 3 illustrating the main capabilities of FHD.

2. The Language (FHD), Sequential Constructs and Symbolic Interpretation

The language FHD is an enhancement of the FP language defined in [17]. FHD can be used to specify synchronous sequential networks. A sequence that is produced sequentially by a synchronous sequential machine can also be produced spatially by a combinational iterative network [12, 7, 5]. Thus, a synchronous sequential system can be specified in FHD using its spatial equivalent - combinational iterative network.

In order to transform the FHD expressions from the behavioral domain to its structural domain, a symbolic interpreter is used. In this interpretation the functions operate on objects which are values or symbols to produce the logic diagram or a net-list. To provide information on the performance parameters, the symbolic interpreter evaluates a number of attributes associated with each primitive function. These attributes represent implementation characteristics such as propagation delay and number of logic levels.

The transformation of FHD expressions from the behavioral domain to the structural domain requires an instantiation of the specification with an object of given dimensions. For example, the description of a multiplexer can be used for any size multiplexer. That is, a 4-input or a 32-input multiplexer have the same description, and for the actual structural realization the size of the input object must be known.

We now consider the use of objects, functions and functional forms of FHD in the structural domain.

Objects In the symbolic domain, objects are associated with both symbols and values; functions operate on objects to generate new symbols, except in the case of predicates. Since predicates are used to control the flow of data, they operate only on values. Thus, in general

each atom must contain a symbol and a value. In order to obtain design characteristics, the values of various attributes are passed along with each object so that each function can update these values depending on its characteristics.

In general, an atom in the symbolic domain has the following form:

(symbol-or-name value optional-list-of-attribute-values)

Currently the symbolic interpreter supports only two types of attributes: the propagation delay(D), and the number of logic levels(L). For example, atom "(MUX-IN1 1 25 3)" can be interpreted as a wire or connection called "MUXIN1" with value of 1. The delay of the corresponding signal is 25 units of time, and the signal has passed through 3 levels of logic. It should be noted that a predicate like atom returns true token "(DUMMY 1 0 0)" when applied to an object like "(MUXIN1 1 10 2)", since this object is an atom in the structural domain.

Functions In the structural domain, functions operate on symbols, values and attributes. For example, or function applied to ((IN1 1 5 1)(IN2 0 9 2)) will produce an atom (WIRE.01 1 18 3). Three categories of functions that appear in FHD must be considered. First, there are the functions that perform basic boolean operations such as and, or, xor, nand, nor, and not. These functions are mapped directly to the corresponding logic gates. Second category contains basic interconnection functions such as "select" and "distribute left". These functions never create new atoms and their effect is independent of the value of their input atoms. They merely rearrange the atoms within an FP object, possibly leaving some out and replicating others [18]. Third, there are functions that are introduced for ease of describing algorithms. Examples are length, atom, null, and predicates. Predicates usually have dual purpose. They are used sometimes to manage the flow of control and ease the description of algorithms, while in other instances a predicate is mapped directly to low-level implementation. In FHD symbolic interpreter, conditional constructs and predicates are used to control the flow of data and ease of algorithm description solely. For functions like length there exists no mapping to a lower level.

The boolean functions include time delay and logic level attributes. Each time a boolean operator is applied, the corresponding delay and logic level attributes are updated as follows:

$$D_{out} = \max(D_1, D_2) + D_f$$

$$L_{out} = \max(L_1, L_2) + 1$$

where D_1 and D_2 denote the time delay of the inputs, L_1 and L_2 represent the logic level of the inputs, and D_f is the propagation delay of the operator. The equations are interpreted in the worst-case sense: the time delay attribute of the output signal is equal to maximum time delay of the inputs plus D_f the time delay of the function (gate). The logic level attribute of the output signal is equal to maximum logic level attributes of the inputs plus one. The following is the output of the symbolic inter-

preter for applying the function HALFADDER to the input "((A0 1 0 0)(B1 1 0 0))" which illustrates how the boolean functions operate on attributes.

```
> 1 HALFADDER.841
> 1 ((A0 1 0 0)(B1 1 0 0))
> 2 AND.842
> 2 (A0 1 0 0)(B1 1 0 0)
> 2 (WIRE.843 1 9 1)
> 2 XOR.844
> 2 (A0 1 0 0)(B1 1 0 0)
> 2 (WIRE.845 0 16 1)
> 1 ((WIRE.843 1 9 1)(WIRE.845 0 16 1))
```

The number following ">" is the level of nested function calls. Each function occupies three lines. The first line has the function name with a number appended at the end, to make a unique function name. The second line lists the inputs, and the third line has the list of outputs. The function HALFADDER.841 has a worst case propagation delay of 16 units of time and has one logic level (as shown by atom WIRE.845). The gate delay of and primitive is 9 units of time and the gate delay of xor gate is 16 units (Mead and Conway [14] timing model is considered).

Functional Forms We now discuss the implementation of functional forms of FHDL in the structural domain. The structure shown for each functional form is generated by the symbolic interpreter and a graphic interpreter [18].

a) Composition Functional Form

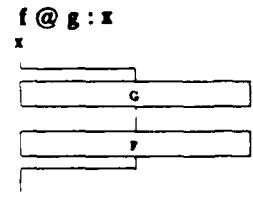


Figure 1. Interpretation of Composition

b) Construction Functional Form

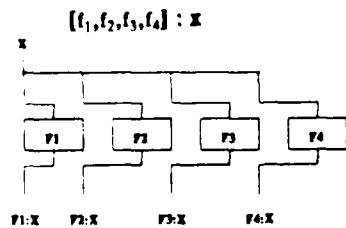


Figure 2. Interpretation of Construction

c) Constant Functional Form

$\%c : x = (name\ x\ 0\ 0\ 0)$

d) Right-Insert Functional Form

$!f : x\ where\ x = (x_1\ x_2\ x_3\ x_4\ x_5)$

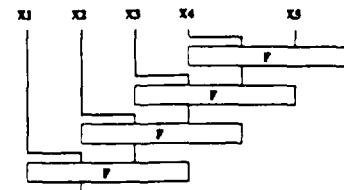


Figure 3. Interpretation of Right-Insert

e) Left-Insert Functional Form

$\backslash f : x\ where\ x = (x_1\ x_2\ x_3\ x_4\ x_5)$

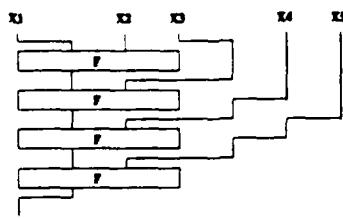


Figure 4. Interpretation of Left-Insert

f) Tree-Insert Functional Form

$\sharp f : x\ where\ x = (x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7)$

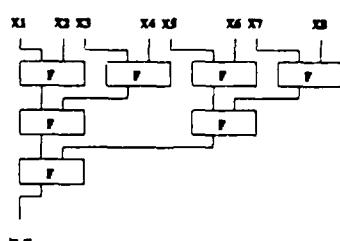


Figure 5. Interpretation of Tree-Insert

Apply-to-All Functional Forms

Two types of *apply-to-all* functional forms are provided for interfacing the sequential functions to the combinational ones and for specifying algorithms in general. These forms are equivalent at the behavioral level, while they differ in the structural domain. This distinction is caused by modeling the sequential systems by iterative networks in the behavioral domain.

a) Space-Apply-to-All Functional Form(&)

$\&f : x = (f(x_1, f(x_2, \dots, f(x_n)),$
where x is a spatial sequence of elements.

Space-apply-to-all functional form at the structural level will be mapped to n copies of function f . The function $\&f$ operates on all elements of the input object concurrently and produces the results simultaneously. This form is equivalent to *apply-to-all* functional form defined by Backus [1].

$\&f : x\ where\ x = (x_1\ x_2\ x_3\ x_4\ x_5)$

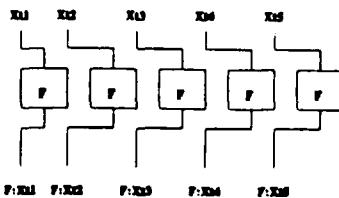


Figure 6. Interpretation of $\&$

b) *Time-Apply-to-All Functional Form(s)*

$\$f x == (f x_{t_1}, f x_{t_2}, \dots, f x_{t_n})$,
where x is a time sequence of elements.

At the structural level *time-apply-to-all* is mapped to one copy of function f . Input x is a time object because the implementation of $\$f$ implies that the elements of the input are applied to f one at a time. In other words, x_{t_i} is the input at time t_1 , x_{t_2} is the input at time t_2 , and so on. In the structural domain, the *time-apply-to-all* functional form only operates on one element of the input at a time (the symbolic interpreter uses the first element of the input objects). *Time-apply-to-all* consumes an input generated by a sequential system and produces an output to be used by a sequential system.

$\$f : x$ where $x = (x_1 \ x_2 \ x_3 \ x_4 \ x_5)$

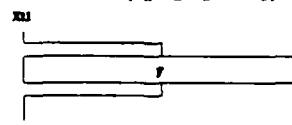


Figure 7. Interpretation of $\$$

We now discuss how these functional forms are used to describe sequential systems.

Sequential Functional Forms

The abstract model of a synchronous sequential system is a finite state machine where x is the input and z is the output, while y is the present state and Y is the next state. The following three functional forms are provided as the implementation of finite state machines. The previous argument used to distinguish between the time and space domain implementations applies directly to these functional forms. All of the following functional forms are implemented in time domain and applied to time objects.

a) *Sequence Functional Form*

$(seq init seqfunc g seqend) x ==$
 $(g:(init x, z_1), g:(g:(init x, z_1), z_2), \dots)$,
where x is a time object.

Keywords seq , $seqfunc$, and $seqend$ act as delimiters. The functions $init$ and g are the initialization and state-transition functions. Sequence functional form describes a finite state machine where the output vector z is the same as the present state y .

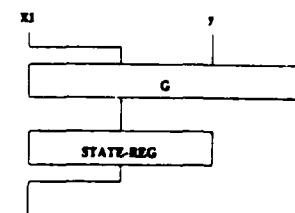


Figure 8. Structural Interpretation of Sequence Functional Form

The following example of a bit-serial adder illustrates the use of *sequence* functional form.

Bit-serial adder using the sequence functional form

```
defun seqadder ()      # OUTPUT ((C_0 S_0) ... (C_{n-1} S_{n-1}))
  seq (%0,%0)      # C_{-1} = 0; S_{-1} = 0
    seqfunc fulladder @
      spndl @ {1@1,2}
    seqend
  enddef      # INPUT ((A_0 B_0) ... (A_{n-1} B_{n-1}))

defun cpnl ()      # OUTPUT ((S_0 S_1 ... S_{n-1}) C_{n-1})
  {S2,
  1@{last} @      # select Sums
  seqadder        # select C_{n-1}
  enddef      # INPUT ((A_0 B_0) ... (A_{n-1} B_{n-1}))
```

Note that in FP programs functions are applied from right to left.

b) *Mealy Functional Form*

$(mealy init meout h menext g meend) : x ==$
 $(h:(init x, z_1), h:(g:(init x, z_1), z_2),$
 $h:(g:(g:(init x, z_1), z_2), z_3), \dots)$
where x is a time object.

The keywords $mealy$, $meout$, $menext$, and $meend$ are FHDL delimiters. The functions $init$, h , and g are the initialization, output, and state-transition functions, respectively. In a Mealy machine, the output depends on both the present state and the input. Function $init$ provides an initial value of the state register.

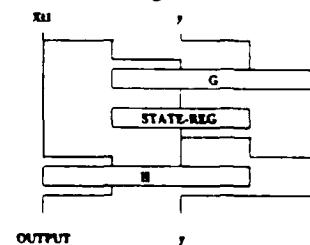


Figure 9. Structural Interpretation of Mealy Functional Form

In the following example a bit-serial adder is specified using the *mealy* functional form.

Bit-serial adder using Mealy functional form

```

defun mealyadder () # OUTPUT ((C0 S0)... (Cn-1 Sn-1))
  mealy [%0,%0] # C-1 = 0; S-1 = 0
  moexit 1 # pass Carry and Sum
  moentry fulladder @
    apndl @ [1@1,2]
  moend
enddef
# ((A0 B0)(A1 B1) ... (An-1 Bn-1))

defun cpa3 () # OUTPUT ((S0 S1 ... Sn-1) Cn-1)
  [S2,
  1@last] @
    # select Sums
    # select Cn-1
  mealyadder
# INPUT ((A0 B0)... (An-1 Bn-1))

```

c) Moore Functional Form

```

(moore init moout h moexit g moend)x ==
(h:(init:x),h:(g:(init:x,x0)),
h:(g:(g:(init:x,x1),x2)),...))
where x is a time object.

```

The keywords *moore*, *moout*, *moexit*, and *moend* are FHDL delimiters. The functions *init*, *h*, and *g* are the initialization, the output and the state-transition functions, respectively. In Moore machine the output depends on the present state only.

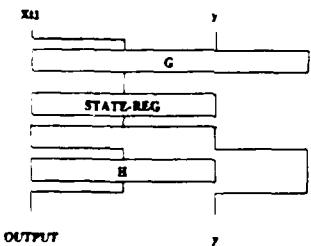


Figure 10. Structural Interpretation of Moore Functional Form

The bit-serial adder is now specified using the *moore* functional form.

Bit-serial adder using Moore functional form

```

defun mooreadder () # OUTPUT ((C0 S0)... (Cn-1 Sn-1))
  moore [%0,%0] # C-1 = 0; S-1 = 0
  moexit id # pass Carry and Sum
  moentry fulladder @
    apndl @
    [1@1,2]
  moend
enddef
# ((A0 B0)(A1 B1)... (An-1 Bn-1))

defun cpa2 () # OUTPUT ((S0 S1 ... Sn-1) Cn-1)
  [S2,
  1@last] @
    # select Sums
    # select Cn-1
  mooreadder
# INPUT ((A0 B0)... (An-1 Bn-1))

```

3. Example

We now illustrate the use of apply-to-all and sequential functional forms by considering the specification and implementation of a multi-operand carry-save adder. Multi-operand carry-save adder uses a carry-save logic with feedback in order to perform additions. The final partial-sum and the carries are passed to a carry-propagate adder to generate the final result. A high-level logic schematic of the adder is shown in Figure 11. The following is an FHDL description of the multi-operand carry-save adder.

```

defun fulladder () # (Ci+1 Si)
  [(1 or 2),3] @
  apndl @ [1, halfadder@[2,3]] @
  apndr @ [halfadder@[1,2],3]
enddef
# (Ai Bi Ci)

defun halfadder () # (Ci+1 Si)
  [and,xor]
enddef
# (Ai Bi)
defun initcsa () # initial value of state
  [(& [%0,%0],%0) @ 1
enddef

defun addall () # ((X0 S0-1 0)...(Xn-1 Sn-1 Rn-1-1))
  & apndr @ trans @ [1@dr@1,2]
enddef
# ((S0-1 0)...(Sn-1-1 Rn-1-1) Cn-1)
# (X0 ... Xn-1))

defun rearrangeoutput () # ((0 S0)(C0 S1)... (Cn-2 Sn-1))
  pair @ apndl @ [%0,id] @ concat
  @ & reverse
enddef
# ((C0 S0)(C1 S1)... (Cn-1 Sn-1))

defun csa () # ((0 S0)... (Rn-1-1 Sn-1) Cn)
  [tr,1@last] @
  rearrangeoutput @
  & fulladder
  @ addall
enddef
# (((0 S0-1)...(Rn-1-1 Sn-1-1) Cn-1)
# (X0 ... Xn-1))

defun multiopcsa () # (((0 S0)...(Rn-1-1 Sn-1)) Cn)
  # ...
  # ((0 S0)...(Rn-1-1 Sn-1)) Cn)
  seq initcsa seqfunc csa seqend
enddef
# ((A01...An-11)... (A0m...An-1m))

defun multiopadder () # ((Pn-11...P01 Cn)...
  # (Pn-1m...P0m))
  (apndr @ [cpa@1,2]) @ last
  @ $([apndl @ [W[%0,%0],1,2]])
  @ multiopcsa
enddef
# ((A01...An-11)... (A0m...An-1m))

```

The function *multiopcsa* specifies the carry-save adder. The function *multiopadder* connects the carry-save adder

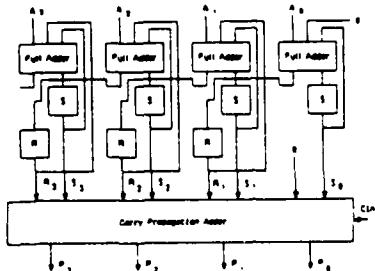


Figure 11. A Multi-operand Carry-save Adder

(the sequential system) to the carry propagation adder (the combinational circuit). The important aspect of this example is the use of the sequential functional form in *cfa* function and the use of *time-apply-to-all* functional form in function *multiopadder* to interface between the sequential part and the combinational one.

Figure 12, 13, and 14 show the logic diagram of functions *multiopadder*, *multiopcsa*, and *cfa* respectively.

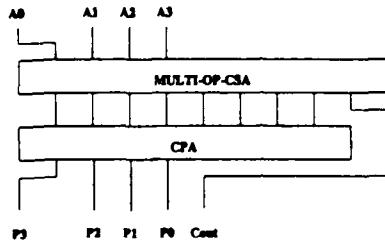


Figure 12. Logic Diagram of Multi-Operand-Adder

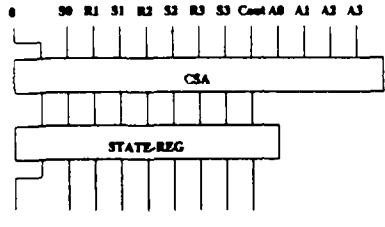


Figure 13. Logic Diagram of Sequential CSA

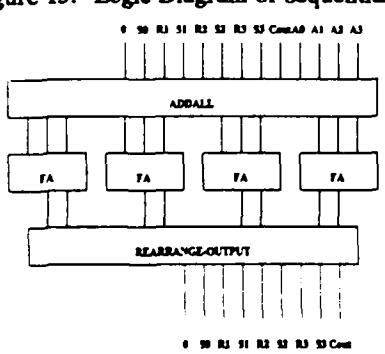


Figure 14. Logic Diagram of a CSA

For carry propagate adder(CPA), function *cfa*, any of the specification in Section 5.2 can be used. A more detailed discussion is given in [15]. As mentioned earlier, two different tools are used in this work. First, the functional interpreter provides a functional simulation of the

FHDL specification. Second, the symbolic interpreter is used to transform the specification from the behavioral domain to the structural domain. The following is a sample of the functional simulation of *multiopadder* and *multiopcsa* functions.

multiopcsa:((0 0 0 0)(0 0 0 1)(0 1 1 1)(0 0 1 0))

((((0 0) (0 0) (0 0) (0 0)) 0) (((0 0) (0 0) (0 0) (0 1)) 0)
(((0 0) (0 1) (1 1) (0 0)) 0) (((0 0) (1 1) (0 1) (0 0)) 0))

multiopadder:((0 0 0 0)(0 0 0 1)(0 1 1 1)(0 0 1 0))

(1 0 1 0 0)

The simulation of *multiopadder* in the structural domain using the symbolic interpreter is given below. Only the high-level modules are shown below.

```
> 1 MULTIOPPADDER.408
> 1 (((IN11 1 0 0)(IN12 1 0 0)(IN13 1 0 0)(IN14 1 0 0))((IN21 1 0 0)
  (IN22 1 0 0)(IN23 1 0 0)(IN24 1 0 0)))

> 2 MULTIOPCSA.409
> 2 (((IN11 1 0 0)(IN12 1 0 0)(IN13 1 0 0)(IN14 1 0 0))((IN21 1 0 0)
  (IN22 1 0 0)(IN23 1 0 0)(IN24 1 0 0)))

> 3 CSA.419
> 3 (((((CONST 410 0 0 0)(CONST 411 0 0 0))((CONST 412 0 0 0)
  (CONST 413 0 0 0))((CONST 414 0 0 0)(CONST 415 0 0 0))
  ((CONST 416 0 0 0)(CONST 417 0 0 0))((CONST 418 0 0 0)))
  ((IN11 1 0 0)(IN12 1 0 0)(IN13 1 0 0)(IN14 1 0 0)))
> 3 (((((CONST 474 0 0 0)(WIRE 431 1 32 2))((WIRE 433 0 34 3)
  (WIRE 444 1 32 2))((WIRE 446 0 34 3)(WIRE 457 1 32 2))
  ((WIRE 459 0 34 3)(WIRE 470 1 32 2)))(WIRE 472 0 34 3)))
> 3 STATE.475
> 3 (((((CONST 474 0 0 0)(WIRE 431 1 32 2))((WIRE 433 0 34 3)
  (WIRE 444 1 32 2))((WIRE 446 0 34 3)(WIRE 457 1 32 2))
  ((WIRE 459 0 34 3)(WIRE 470 1 32 2)))(WIRE 472 0 34 3)))
> 3 (((((CONST 410 0 0 0)(CONST 411 1 0 0))((CONST 412 0 0 0)
  (CONST 413 1 0 0))((CONST 414 0 0 0)(CONST 415 1 0 0))
  ((CONST 416 0 0 0)(CONST 417 1 0 0)))(CONST 418 0 0 0)))
> 2 (((((CONST 410 0 0 0)(CONST 411 1 0 0))((CONST 412 0 0 0)
  (CONST 413 1 0 0))((CONST 414 0 0 0)(CONST 415 1 0 0))
  ((CONST 416 0 0 0)(CONST 417 1 0 0)))(CONST 418 0 0 0))

> 2 CPA.478
> 2 (((((CONST 476 0 0 0)(CONST 477 0 0 0))((CONST 410 0 0 0)
  (CONST 411 1 0 0))((CONST 412 0 0 0)(CONST 413 1 0 0))
  ((CONST 414 0 0 0)(CONST 415 1 0 0))((CONST 416 0 0 0)
  (CONST 417 1 0 0)))
> 2 ((WIRE 490 1 32 2)(WIRE 504 1 50 4)(WIRE 518 1 68 6)
  (WIRE 532 1 86 8))

> 1 ((WIRE 490 1 32 2)(WIRE 504 1 50 4)(WIRE 518 1 68 6)
  (WIRE 532 1 86 8)(CONST 418 0 0 0))
```

The symbolic interpreter does not provide the full simu-

lation of the FHDL specification. The interpreter simulates the specification only to the level necessary to transform the specification to the structural domain. As illustrated above, the function *multiopadder* was applied to the input consisting of two numbers represented with bit-vectors 1111 and 1111. The output of symbolic interpreter was 1111, because the interpreter executes the *multiopcsa* function only once. That is, it adds the first input with the initial value of state register *w*. The initial value of state register is provided by function *initcsa* which is zero.

The maximum clock rate of *multiopadder* is 34 units of time plus the time delay of the state register (i.e., 8 units of time). The delay of *cpa* function is 86 units of time. Thus, for adding 20 numbers about $20 \times (34 + 8) + 86 = 926$ units of time are required.

As illustrated above, FHDL can be used to specify digital systems, to map the specification into a gate level implementation, and to simulate its functional behavior. The use of attributes provides a systematic method for gathering performance characteristics of the design.

4. Conclusion

A functional programming hardware description language (FHDL), based on the Backus's FP, was described. FHDL supports the specification of both combinational and sequential systems. The sequential systems are modeled by equivalent iterative networks at the behavioral level. Then, a symbolic interpreter is used to interpret the specifications automatically at the structural level, which presently consists of gates, registers and interconnections. Characteristic attributes are introduced in a general manner so that the information about system performance and design parameters can be extracted. The two implemented attributes are the delay and the number of levels.

Acknowledgements: This research has been supported in part by the Office of Naval Research Contract N00014-83-K-0493 and the MICRO-Rockwell International Grant. The authors are indebted to D.R. Patel, M.D.F. Schlag, S.L. Lu and J. Worley for their contributions to the FP environment at UCLA which made this work possible.

References

- [1] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communication ACM*, Vol. 21, No. 8, Aug. 1978, pp. 613-641.
- [2] Backus, J., "Function-level Programming," *IEEE Spectrum*, Vol. 19, No. 8, Aug 1982, pp. 22-27.
- [3] Baden, S. B. and D. R. Patel, "Berkeley FP -- Experiences with a Functional Programming Language," *Proc. of COMPCON*, Spring 1983, pp. 274-277.
- [4] Cardelli, L. and G. Plotkin, "An Algebraic Approach to VLSI Design," in *VLSI 81*, 1981, pp. 173-182.
- [5] Davio, M., J. P. Deschamps, and A. Thyse, in *Digital Systems with Algorithm Implementation*, John Wiley & Sons, 1983.
- [6] Ercegovac, M.D. and T. Lang, "A High-Level Language Approach to Custom Chip Layout Design," *University of California MICRO Project Reports 1982-83*, April 1984.
- [7] Ercegovac, M. D. and T. Lang, in *Digital Systems: Hardware/Firmware Algorithms*, New York, J. Wiley & Sons, 1985.
- [8] Frankel, R. E. and S. W. Smoliar, "Beyond Register Transfer: An Algebraic Approach For Architectural Description," *Proc. of 4th International Conf. on Computer Hardware Description Languages*, Oct. 1979, pp. 1-5.
- [9] Frankel, R. E. and S. W. Smoliar, "Digital Systems as Mathematical Expressions," *Proc. of COMPCON*, Spring 1981, pp. 414-416.
- [10] Gordon, M., "A Model of Register Transfer System with Applications to Microcode and VLSI Correctness," *Tech. Rep. Unpublished*, 1981.
- [11] Johnson, S.D., *Synthesis of Digital Designs from Recursion Equations*, Cambridge, Mass.: The MIT Press, 1984.
- [12] Kohavi, Z., in *Switching and Finite Automata Theory*, McGraw Hill, 1978.
- [13] Lahti, D. O., "Application of a Functional Programming Language," UCLA Dept. of Computer Science, LA, CA, Tech. Rep. Report No. CSD-810403, 1981.
- [14] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Reading, Massachusetts: Addison-Wesley, 1980.
- [15] Meshkinpour, F., "On Specification and Design of Digital Systems Using an Applicative Hardware Description Language," University of California, Los Angeles, Tech. Rep. Master Thesis, March 1984.
- [16] Patel, D., M. Schlag, and M.D. Ercegovac, "vFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms," *UCLA Computer Science Department Quarterly*, January 1985.
- [17] Patel, D. and J. Worley, *The UCLA Functional Programming Language*: UCLA Computer Science Department, 1985, submitted for publication.
- [18] Schlag, M., "Extracting Geometry from FP for VLSI Layout," UCLA Department of Computer Science, Tech. Rep. CSD-840043, Oct. 1984.
- [19] Sequin, C. H., "Managing VLSI Complexity: An Outlook," *Proceedings of IEEE*, Vol. 71, No. 1, Jan. 1983, pp. 149-166.
- [20] Sheeran, M., "muFP, a Language for VLSI Design," *Proc. 1984 ACM Conference on Lisp and Functional Programming*, August 1984, pp. 104-112.

**ARITHMETIC ALGORITHMS FOR OPERANDS ENCODED IN
TWO-DIMENSIONAL LOW-COST ARITHMETIC ERROR CODES***

Algirdas Avizienis

UCLA Computer Science Department
University of California
Los Angeles, CA 90024, USA

Abstract

A generalization of low-cost residue codes into two-dimensional encodings was presented and error detecting and error correcting properties of two dimensional inverse residue codes were discussed previously. This paper presents byte-serial checking, additive inverse (complementation), and addition algorithms for operands encoded in two-dimensional residue and inverse residue codes.

1. Introduction

A general approach to the cost and effectiveness study of low-cost arithmetic error codes has been presented in [AVIZ 71a]. This paper introduced the concepts of *inverse residue codes* and of *multiple arithmetic error codes*. The concept of *repeated use faults* was presented and the effectiveness of various arithmetic codes with respect to both *determinate* and *indeterminate* repeated-use faults was established. An important result was the proof that inverse residue codes can detect the "compensating" determinate repeated-use faults that are not detected by ordinary residue codes. The modulo 15 inverse residue code was applied in the JPL-STAR experimental computer [AVIZ 71b]. Further results on determinate faults were presented in [PARH 73] and [PARH 78]. An extension to signed-digit arithmetic is found in [AVIZ 81]. J. F. Wakerly has analyzed the detectability of unidirectional multiple errors [WAKE 75], and A.M. Usas has demonstrated the advantages of inverse residue codes for multiple unidirectional error detection, when compared to inverse checksum codes [USAS 78]. Bose and Rao have considered unidirectional one-line error correcting codes using a combination of byte parity and residue (not low-cost) encoding (BOSE 80).

A new generalization presented in [AVIZ 83] extended the application of low-cost inverse residue codes into two dimensions: row (byte) and column (line) residues.

* This research has been supported by ONR contract N00014-83-K-0493.

This extension improves the detection of errors, especially of those due to indeterminate faults, and provides certain error-correction capabilities. Of special interest to current VLSI implementations of arithmetic are the advantages offered by two-dimensional inverse residue codes in the detection and correction of errors that affect byte-wide communication paths and processing elements. Such paths are widely used in high-performance array processors, systolic arrays, and for inter-processor communication in large multi-processor systems. Byte-wide processing elements are very suitable for the implementation of large processing arrays [AVIZ 70], [TUNG 70] and variable-precision signed-digit arithmetic [AVIZ 62].

This paper presents the fundamental byte-serial arithmetic algorithms for operands encoded in two-dimensional low-cost inverse residue codes. The algorithms are:

- (a) the line-residue checking algorithm;
- (b) the additive inverse (complementation) algorithm;
- (c) the addition algorithm.

A brief review of the error-detecting and error-correcting properties of 2-D inverse residue codes follows the description of arithmetic algorithms.

2. Model of the Byte-Serial Communication and Computation Path

We consider a *communication and computation path* consisting of b bit lines $(X^0, X^1, \dots, X^{b-1})$. The binary operand X consists of kb bits, processed as k bytes $(X_0, \dots, X_1, \dots, X_{k-1})$ of b bits length each. Figure 1 shows the notation used in this paper.

Two types of low-cost residue encoding are applicable to the operand X :

- (a) *Residue Code*: the k bytes carry an error-

detecting code check byte X_k that represents the modulo 2^b-1 residue X' of the operand X : $X' = (2^b-1)|X$; and the operand is now $k+1$ bytes long. Usually the residue value $X'=0$ is represented by a string of b ones. If the all-zero operand can exist, its residue will be b zeros, unless explicitly disallowed.

(b) *Inverse Residue Code*: the inverse residue byte X_k represents the value X'' that is the (2^b-1) 's complement of X' . It is obtained as $X'' = (2^b-1) - X' = (2^b-1) - (2^b-1)|X$; and the operand is again $k+1$ bytes long. The residue value $X''=0$ is represented by b ones, and the inverse residue X'' in this case is represented by b zeros. The all-zero operand X has an inverse residue code X'' represented by b ones.

To form a two-dimensional residue encoding, one more check line X^b is added to the communication and computation path (Figure 1). The lines X^0, X^1, \dots, X^{b-1} are summed modulo $2^{k+1}-1$ to get the *line-residue* of X . Two classes of 2-D low-cost residue codes can be employed:

(c) *Two-dimensional Residue Code*: the check bits X_j^b of the check line X^b represent the modulo $2^{k+1}-1$ *line-residue* X_L^b :

$$X_L^b = (2^{k+1}-1) \mid \sum_{j=0}^{b-1} X_j^b; \text{ where } X_j^b = \sum_{i=0}^k X_i^j 2^i$$

(d) *Two-dimensional Inverse Residue Code*: the check bits X_j^b on the check line X^b represent the modulo $2^{k+1}-1$ *inverse line-residue* X_L'' :

$$X_L'' = (2^{k+1}-1) - X_L^b$$

It is important to note that the bits $(X_k^{b-1}, \dots, X_0^0)$ of the check byte X_k are treated as the most significant bits of the lines X^{b-1}, \dots, X^0 when the line-residue of X is determined. The line-residue encoding is superimposed on the already encoded operand.

Check line X^0	line X^{b-1}	...	line X^1	...	line X^1	line X^0	byte symbol
x_0^0	x_0^{b-1}	...	x_0^1	...	x_0^1	x_0^0	x_0
\vdots	\vdots		\vdots		\vdots	\vdots	\vdots
x_1^0	x_1^{b-1}	...	x_1^1	...	x_1^1	x_1^0	x_1
\vdots	\vdots		\vdots		\vdots	\vdots	\vdots
x_{k-1}^0	x_{k-1}^{b-1}	...	x_{k-1}^1	...	x_{k-1}^1	x_{k-1}^0	x_{k-1}
x_k^0	x_k^{b-1}	...	x_k^1	...	x_k^1	x_k^0	check byte X_k

Figure 1. Model of the Path and the Operand X

3. A Byte-Serial Line-Residue Checking Algorithm

Given a Two-Dimensional Inverse Residue encoded operand X (as shown in Figure 1), the line-residue checking algorithm requires the computing of the *line check result* $R(L)$:

$$R(L) = (2^{k+1}-1) \mid \sum_{j=0}^b X_j^b; \text{ where } X_j^b = \sum_{i=0}^k X_i^j 2^i$$

An "all-ones" $R(L)$ indicates a valid encoding; all other values of $R(L)$ indicate an error.

In the byte-serial implementation, one byte of X becomes available at a time, with the least significant byte X_0 arriving first. The "line sum" $\sum_{j=0}^b X_j^b$ designated by $\sum(L)$, is computed as the weighted sum of the "count of ones" in each byte:

$$\sum(L) = \sum_{j=0}^b X_j^b = \sum_{i=0}^k (\sum_{j=0}^b X_i^j) 2^i$$

In order to get $R(L)$, the modulo $(2^{k+1}-1)$ residue of $\sum(L)$ must be computed. That requires an "end-around-carry" addition of the "overflow bits" $(S_{k+m}, \dots, S_{k+1})$ of $\sum(L)$ that are in the positions $k+1, \dots, k+m$ of $\sum(L)$. Since a full-length addition delay is unacceptable in high-speed byte-organized computing (e.g., systolic arrays), a fast line-residue checking

algorithm is developed here that requires only a short, m -bit (with $2^m \geq b+1$) addition after $\sum(L)$ has been byte-serially computed.

The speed-up is accomplished by the simultaneous computing of two tentative line sums $\sum(L)$ and $\sum(L)' = \sum(L) + 2^m$. The value of m is determined by the maximum value of the line sum $\sum(L)$. An upper bound for $\sum(L)$ is obtained by assuming all digits X_i^j ($0 \leq i \leq k$; $0 \leq j \leq b$) to have the value "1". (This situation cannot occur for a valid encoding, but could be caused by an error.) In this case,

$$\sum(L)_{\max} = (b+1)(2^{k+1}-1) = b2^{k+1} + (2^{k+1}-1) - b$$

and the "overflow bits" represent the value b with respect to the position $k+1$ of the line sum $\sum(L)$. The value of m is the smallest integer that satisfies the condition:

$$2^m - 1 \geq b \text{ or } 2^m \geq b+1$$

For example, 8-bit bytes ($b=8$) will need $m=4$, regardless of operand length k (in bytes).

After $\sum(L)$ and $\sum(L)' = \sum(L) + 2^m$ have been computed, the m "overflow bits" $(S_{k+m}, \dots, S_{k+1})$ of $\sum(L)$ are added to the m least significant bits (S_{m-1}, \dots, S_0) of $\sum(L)$. The resulting carry-out C_m determines the choice of the bits (S_k, \dots, S_m) :

(a) If $C_m = 0$, (S_k, \dots, S_m) come from $\sum(L)$

(b) If $C_m = 1$, (S_k, \dots, S_m) come from $\sum(L)'$

The "all ones" line check result $(S_i = 1 ; 0 \leq i \leq k)$ indicates the absence of errors; any other line check result is an error indication. The determination whether (S_k, \dots, S_m) are "all ones" is done while these bits are computed. The final step is to test whether the bits (S_{m-1}, \dots, S_0) also are "all ones" after the "end-around" addition of the "overflow bits".

4. The Byte-Serial Additive Inverse Algorithm

The additive inverse of an operand X is formed by obtaining the complement of X . Either "one's" or "two's" complements can be employed; the specifics are discussed in [AVIZ 71a] and [AVIZ 73].

The purpose of this section is to develop the corresponding complementation algorithm for the inverse line-residue X^b . When the "one's" complement \bar{X} of X is formed, the "count of ones" in each byte X_i of X is :

$$\sum_{j=0}^{b-1} X_j^b = b - \sum_{j=0}^{b-1} X_j^0$$

This leads to the relationship for $\sum(\bar{X})$:

$$\sum(\bar{X}) = \sum_{i=0}^k (b - \sum_{j=0}^{b-1} X_j^b) 2^i = b(2^{k+1} - 1) - \sum(X)$$

Taking the line-residue modulo $(2^{k+1} - 1)$, we get:

$$\begin{aligned} (2^{k+1} - 1) | \sum(\bar{X}) &= \\ &= (2^{k+1} - 1) | (0 - [(2^{k+1} - 1) | \sum(X)]) = \\ &= (2^{k+1} - 1) - (2^{k+1} - 1) | \sum(X) \end{aligned}$$

The relationship demonstrates that the line-residue of the "one's" complement \bar{X} is obtained by taking the one's complement $(2^{k+1} - 1) - [(2^{k+1} - 1) | \sum(X)]$ of the line-residue $(2^{k+1} - 1) | \sum(X)$ that was computed for the operand X . The same argument follows for the inverse line-residue.

If the "two's" complement X^0 of X is to be formed, it is considered to be:

$$X^0 = \bar{X} + 2^0$$

In order to get the line-residue of X^0 , the line-residue of 2^0 must be added modulo $2^{k+1} - 1$ to the line-residue of X . For inverse line-residue encoding, this means the addition of $C_0^0 = 1$, as described in the next section.

5. The Byte-Serial Addition Algorithm

We consider the byte-serial addition of two operands X and Y , each k bits long, to get the sum $Z = X + Y$. The addition is modulo $2^b - 1$ ("one's" complement), or modulo 2^b ("two's" complement).

The check byte Z_k is obtained by adding the check bytes X_k and Y_k modulo $2^b - 1$. If "two's" complement is used, a "correction signal" input needs to be used, as defined in [AVIZ 73].

An algorithm to generate the inverse line-residue for Z from the inverse line-residues of X and Y is developed here. As first developed by Garner [GARN 58], the carries generated during the addition of X and Y need to be employed in the calculation of the inverse line-residue for Z .

The "count of ones" (designated by $\alpha(Z_i)$) in each byte Z_i ($0 \leq i \leq k-1$) of Z is:

$$\sum_{j=0}^{b-1} Z_j^i = \sum_{j=0}^{b-1} (X_j^i + Y_j^i) - (\sum_{j=1}^{b-1} C_j^i) - 2C_k^i + C_0^i ;$$

$$\text{or: } \alpha(Z_i) = \alpha(X_i) + \alpha(Y_i) - \alpha(\text{internal } C_i) - 2C_k^i + C_0^i ;$$

where C_j^i is the carry into the j -th position of the sum byte Z_i , and $C_{i+1}^0 = C_i^b$ for $0 \leq i \leq k-2$. For "one's" complement addition of X and Y , we also have $C_{k-1}^b = C_0^0$.

The above leads to an expression for $\sum(Z)$ when

$$\sum(Z) = \sum_{i=0}^{k-1} \alpha(Z_i) 2^i + \alpha(Z_k) 2^k ;$$

or

$$\begin{aligned} \sum(Z) &= \sum^*(X) + \sum^*(Y) - \sum^*(\text{internal } C) - 2^k C_{k-1}^b + \\ &\quad + C_0^0 + 2^k [\alpha(X_k) + \alpha(Y_k) - \alpha(C_k)] \end{aligned}$$

The count $\alpha(C_k)$ is the total count of carries C_j^i for $1 \leq j \leq b$, since $C_k^0 = C_k^b$ in the modulo $2^b - 1$ addition of the check bytes X_k and Y_k ; i.e.,:

$$\alpha(C_k) = \sum_{j=1}^b C_j^i = \alpha(\text{int } C_k) + C_k^b$$

Two cases need to be discussed separately:

(a) "One's" complement (modulo $2^b - 1$) addition of X and Y ;

(b) "Two's" complement (modulo 2^b) addition of X and Y .

For "one's" complement, $C_{k-1}^b = C_0^0$ is the "end-around-carry", and the expression for $\sum(Z)$ is :

$$\sum(Z) = \sum'(X) + \sum'(Y) - \sum'(\text{inv. } C) - 2^k C_{k-1}^b + C_k^b + 2^k a(X_k) + 2^k a(Y_k) - 2^k a(\text{inv. } C_k) - 2^k C_k^b$$

The expression reduces to:

$$\sum(Z) = \sum(X) + \sum(Y) - [\sum(\text{inv. } C) + 2^k(C_k^b + C_{k-1}^b) - C_{k-1}^b]$$

Taking the line-residue modulo $A = 2^{k+1} - 1$ of $\sum(Z)$, we get:

$$A \mid \sum(Z) = A \mid [A \mid \sum(X) + A \mid \sum(Y) - A \mid [\sum(\text{inv. } C) + 2^k(C_k^b + C_{k-1}^b) - C_{k-1}^b]]$$

This relationship shows that the line-residue of Z can be predicted from the line-residues of X and Y , as well as a line-residue computed from the internal carries formed during the summation of X and Y . The two "end-around" carries C_{k-1}^b and C_k^b also need to be included in the calculation.

Common-mode errors can occur if the carries are incorrectly determined. To avoid such errors, *separate* and *independent* carry-forming circuits need to be employed to form the carries for line-residue determination.

For two's complement, the "correction signal" C_{k-1}^b must be added (modulo $2^b - 1$ to the modulo $2^{k+1} - 1$ sum of inverse residue check bytes X_k and Y_k [AVIZ 73].

The expression for $\sum(Z)$ is now:

$$\sum(Z) = \sum'(X) + \sum'(Y) - \sum'(\text{inv. } C) - 2^k C_{k-1}^b + C_0^0 + 2^k a(X_k) + 2^k a(Y_k) - 2^k a(\text{inv. } C_k) - 2^k C_k^b + 2^k C_{k-1}^b$$

The expression is reduced to:

$$\sum(Z) = \sum(X) + \sum(Y) - [\sum(\text{inv. } C) + 2^k C_k^b - C_0^0]$$

where $C_0^0 = 1$ only exists if one of the two operands is being complemented (in "two's" complement) simultaneously with the addition. Once again, we take the line-residue modulo $A = 2^{k+1} - 1$ of $\sum(Z)$ as follows:

$$A \mid \sum(Z) = A \mid [A \mid \sum(X) + A \mid \sum(Y) - A \mid [\sum(\text{inv. } C) + 2^k C_k^b - C_0^0]]$$

The difference between "two's" and "one's" complement cases is quite small with respect to computing the line-residue $(2^{k+1} - 1) \mid \sum(Z)$.

In practical implementation of byte-serial arithmetic the "two's" complement addition (and subtraction) is strongly preferable because there is no "end-around-carry" that requires either a second addition or the generation of two "tentative" sums - with and without the end-around-carry.

6. Detection of Unidirectional and Bidirectional Errors

In this and the following sections 7 and 8, the error-detecting and error-correcting properties of the two-dimensional codes that were first presented in [AVIZ 83] are reviewed, illustrated, and extended to two and three adjacent lines.

Given a modulo $2^b - 1$ inverse residue code, the undetectable unidirectional errors are those that have error values E congruent to zero modulo $2^b - 1$, where

$$E = \sum_{j=0}^{b-1} \left(\sum_{i=0}^k E_i \right) 2^j$$

All other unidirectional errors will be detected; however, there are no error correction properties.

One bit-line determinate ("stuck line") faults that cause unidirectional errors will always be detected as long as the condition:

$$(k+1) < (2^b - 1)$$

is satisfied. For two adjacent "stuck lines," the condition is:

$$3(k+1) < (2^b - 1)$$

For m adjacent "stuck lines," the condition is:

$$(2^m - 1)(k+1) < (2^b - 1)$$

For the purpose of this discussion, lines 0 and $b-1$ are considered adjacent.

The *PM* (pattern miss) [AVIZ 81] percentages for "stuck line" faults remain very low after the left side of the inequalities above exceeds the limit that guarantees *PM* percentage of 0%. For example, for one "stuck line," when $k+1 = 2^b - 1$ is reached, we have:

$$PM(\text{Inv. Residue}) = 100/(2^{k+1})$$

since only one of the 2^{k+1} possible error patterns on the "stuck line" (all zeros - all ones, or vice versa) goes undetected. The situation is not as favorable with "stuck byte" faults, as discussed next.

There is one undetectable one-byte unidirectional error; it results when an all-zero byte X_i is changed to an all-ones byte, or vice versa. The *PM* percentage for this "stuck byte" fault is $(100/2^b)\%$. Introduction of byte parity bits will detect only one of the two (stuck-on-one and stuck-on-zero) "stuck bytes"; the other one remains undetectable.

The "stuck byte" detection problem is fully solved by the use of two-dimensional inverse residue encoding. There is one additional check bit X_i^b for each byte X_i ($i = 0, \dots, k$). The check bits (X_k^b, \dots, X_0^b) represent

the modulo $2^{k+1}-1$ inverse line-residue Y^* of the operand X that is now interpreted as b lines X^j ($j=0, \dots, b-1$) of $k+1$ bits length each. It is evident that every "stuck byte" now will be detected by the use of Y^* as long as the condition:

$$(b+1) < (2^{k+1}-1)$$

is satisfied. For two adjacent "stuck bytes," the condition is:

$$3(b+1) < (2^{k+1}-1);$$

for p adjacent "stuck bytes" it is:

$$(2^p-1)(b+1) < (2^{k+1}-1)$$

The bytes X_0 and X_k are considered adjacent in this analysis.

The two-dimensional inverse residue is clearly superior to the byte-parity encoding, since the "stuck byte" condition subsumes all other possible error patterns (double, quadruple, etc.) in the byte, while all "even error" patterns go undetected when byte parity is the only form of encoding.

In general, the remaining undetectable errors in the operand X are those that are missed by both checks: modulo 2^b-1 over the bytes (not including the check line bits X_i^b), and modulo $2^{k+1}-1$ over the lines, with the check byte bits X_i^b included in each line j . Most unidirectional errors are detectable; furthermore, the detection of bidirectional errors is significantly improved, as discussed below.

It has been noted that low-cost inverse residue codes are considerably less effective in detecting bidirectional errors due to indeterminate repeated-use faults [AVIZ 71a]. The addition of the line-residue (i.e., the second dimension of encoding) allows the detection of all bidirectional errors that affect a single line, as well as all bidirectional double errors affecting any two bits of the operand X . The double, quadruple, and other even "half-and-half" bidirectional errors on one line that were undetected by the byte check are now detected by the line check, while those in one byte are detected by the byte check.

The remaining undetectable bidirectional errors are those that are simultaneously undetectable by the byte check and the line check. An illustration is the quadruple error that changes Z to Z^* as shown below:

$$Z = \begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix} \Rightarrow Z^* = \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

Here an even number of opposite-direction changes occurs simultaneously in the bytes and lines of the operand X . In general, all quadruple errors of this type (at four corners of a rectangle of bits within the operand X) are undetectable.

7. Correction of Single-Bit and Unidirectional Single-Line Errors

The introduction of the inverse line-residue Y^* also makes single-bit error correction possible. As shown in [AVIZ 71a], the low-cost inverse residue codes have the "partial error location" property. Therefore a single-bit error value $E_i^j = \pm 1$ ($0 \leq j \leq b-1$; $0 \leq i \leq k$) will produce a unique indication for line j in the modulo 2^b-1 check and for the byte i in the modulo $2^{k+1}-1$ check, making a correction of E_i^j possible in the operand X . The single-bit error $E_i^b = \pm 1$ that occurs in the check line b will produce the indication for byte i ($0 \leq i \leq k$) in the modulo $2^{k+1}-1$ check, but no error indication at all in the modulo 2^b-1 check, since it does not include the bits of the check line. Correction of E_i^b is therefore possible.

The correction property can be extended to most unidirectional single-line errors as follows. If we assume a determinate single-line fault on line j , the error values $E(j)$ will fall into the range:

$$-2^j \sum_{i=0}^k E_i^j \leq E(j) \leq 2^j \sum_{i=0}^k E_i^j$$

The positive values will be due to a stuck-on-one (s-o-1) and negative values — due to a stuck-on-zero (s-o-0). The actual byte check results will assume the values

$C(j) = (2^b-1) \mid E(j)$, and as long as $(k+1) < (2^b-1)$ holds, all error values due to a s-o-1 fault will be detectable and have a unique byte check result $C(j)$ in the range

$$0 \leq C_1(j) \leq (2^b-1) \mid (k+1)2^j$$

Similarly, the error values due to a s-o-0 fault will have the byte check result in the range:

$$0 \leq C_0(j) \leq (2^b-1) \mid (-2^j)(k+1)$$

However, many other error patterns (on two or more lines) can produce the same values of check results, and error correction is not possible with the byte residue encoding alone.

To obtain single-line unidirectional error correction, we use the additional information provided by the line check result obtained from the inverse line-residue encoding. Given a byte check result $C_1(j)$ discussed above, we find its value to be N , represented by b bits (N_{b-1}, \dots, N_0).

First we form the hypothesis that N is due to a single-line stuck-on-one determinate fault on line j ($0 \leq j \leq b-1$). If the fault is in line $j=0$, then $N(0) = N$ error bits $E_i^0 = 1$ in line 0 will produce the byte check result N . We determine the numbers $N(j)$ of error bits $E_i^j = 1$ on lines $j=1, \dots, b-1$ respectively that would be needed to produce the byte check result N by end-

around shifting N to the right $b-1$ times. The shifts will produce the numbers $N(1), \dots, N(b-1)$ in succession.

The number of error bits $E_j = 1$ (due to a stuck-on-zero line) that would be needed to produce $N(j)$ for any $0 \leq j \leq b-1$ is given by $(2^b - 1) - N(j)$, that is, the "one's complement" of $N(j)$. All values of $N(j)$ and $(2^b - 1) - N(j)$ that are greater than $k+1$ are discarded as impossible solutions.

To test the hypothesis that a given byte check result N is due to a single-line determinate fault, we use the line check result

$$R = (2^{k+1} - 1) \mid \sum_{j=0}^b \left(\sum_{i=0}^k X_j^i 2^i \right)$$

This result will contain $N(j)$ digits $R_i = 1$ ($0 \leq i \leq k+1$) if there is a single-line determinate (stuck-on-one) fault in the line j . The presence of each $R_i = 1$ indicates that the digits X_j^i should be corrected by the 1-0 change.

The line check result R will contain $(2^b - 1) - N(j)$ digits $R_i = 0$ ($0 \leq i \leq k+1$) if there is a single-line determinate (stuck-on-zero) fault in the line j . The presence of each $R_i = 0$ indicates that the digit X_j^i should be corrected by the 0-1 change.

Example 1: Line Correction

Consider an operand X with seven bytes ($k = 7$) of 4 bits each ($b = 4$). Inverse-residue encoding is used for the bytes (modulo $2^b - 1 = 15$) and for the lines (modulo $2^{k+1} - 1 = 255$). The encoded operand (following Figure 1) is shown below:

check line	line 3	line 2	line 1	line 0	
1	0	1	0	0	byte 0
0	0	0	1	1	byte 1
0	1	0	0	1	byte 2
0	0	0	0	0	byte 3
0	1	0	1	1	byte 4
1	0	0	1	0	byte 5
0	1	0	1	0	byte 6
0	0	1	1	0	check byte 7

The byte check result (modulo 15) is $N = 1111$, and the line check result (modulo 255) is $R = 11111111$. No errors are indicated.

Now assume a stuck-on-one line 2 and set all digits in line 2 to one. The new byte check result is $N = 1001$. The single-line determinate fault possibilities are:

Stuck-on-One	Stuck-on-Zero
$N(0) = 1001 = 9$	$15 - N(0) = 6$
$N(1) = 1100 = 12$	$15 - N(1) = 3$
$N(2) = 0110 = 6$	$15 - N(2) = 9$
$N(3) = 0011 = 3$	$15 - N(3) = 12$

The values greater than $k+1=8$ are discarded, and the remaining possibilities are: line 2 (6 errors) or line 3 (3 errors) stuck-on-one, and line 0 (6 errors) or line 1 (3 errors) stuck-on-zero.

The new line check result is

$$R = (R_7, \dots, R_0) = 01111110$$

The six ones in R indicate that the "line 2 stuck-on-one" hypothesis is valid, and the corresponding six positions in line 2 are corrected by setting them to zero.

The single-line, unidirectional error correction algorithm can not be completed only in the cases in which two conditions occur simultaneously:

- (a) More than one line is indicated by the occurrence of identical values of $N(j)$ or of $15 - N(j)$ for two or more lines j of X .
- (b) The correction pattern indicated by the line check result R is actually *applicable* to more than one line j of the operand X , i.e., the lines have all zeros (or all ones) in the positions to be corrected.

Example 2 below illustrates condition (2); the subsequent discussion deals with condition (b).

Example 2: Correction Ambiguity

Now assume that line 1 is stuck-on-zero. The byte check result is $N = 0101$, and the possibilities are:

Stuck-on-One	Stuck-on-Zero
$N(0) = 0101 = 5$	$15 - N(0) = 10$
$N(1) = 1010 = 10$	$15 - N(1) = 5$
$N(2) = 0101 = 5$	$15 - N(2) = 10$
$N(3) = 1010 = 10$	$15 - N(3) = 5$

The remaining possibilities all point to five errors. The modulo 255 line check result is

$$R = 00001101$$

The five zeros in R (positions 7,6,5,4,1) indicate a stuck-on-zero on line 1 or line 3. To resolve the ambiguity, we find that line 3 already has "1" digits in positions 6 and 4, and cannot be corrected there; therefore the stuck line must be line 1.

It is possible that both potential corrections could be carried out in Example 2 above; that is, both line 1 and

line 3 could have zeros in positions 7,6,5,4,1. In such a case, the error has been detected, but a correction is not possible, since both conditions (a) and (b) occur simultaneously.

8. Two-Line and Three-Line Errors

A more critical case than the ambiguity discussed above would be that of a mis-correction, in which the restored pattern would differ from the original one, such as in the case of triple errors encountered by the Hamming SEC/DED code.

A mis-correction for two-dimensional inverse residue codes will occur if the bit pattern of the operand X changes in more than one line, but *both* the byte check result value N and the line check result value R remain the same as for a single-line error. This will happen when:

- (a) the byte check result is altered by $\pm c(2^b-1)$
- (b) the line check result is altered by $\pm c(2^{k+1}-1)$
- (c) both (a) and (b) occur simultaneously.

In cases (a) and (b) the other check result remains unchanged.

It is readily shown that a mis-correction cannot occur if only two adjacent lines (or bytes) are affected by the fault; the detection is guaranteed in all cases. When three adjacent lines (or bytes) are affected, a mis-correction can occur. The byte check result will be altered by $\pm(2^b-1)$ when the following changes are imposed on a correctable unidirectional single-line error pattern:

- (a) two error bits from line j are moved one line to the right, causing a net change in N of $2(2^{b-1}) \cdot 2 = 2^b \cdot 2$;
- (b) one error bit from line j is moved one line to the left, causing a net change in N of $2 \cdot 1 = 1$.

The total change in N is then $(2^b \cdot 2) + 1 = 2^b \cdot 3$, and it will lead to a mis-correction if the following two conditions are satisfied:

- (1) there are no further error changes, and
- (2) the positions in line j that would be mis-corrected actually *do* contain correctable bit values.

An example of the conditions under which a mis-correction will occur is shown in Example 3 below.

Example 3: Conditions for Mis-Correction

Consider the encoded operand below (same format as in Example 1). Without changes, both $N=1111$ and $R=11111111$ are obtained.

check line	line 3	line 2	line 1	line 0	
1	0	1	0	0	byte 0
1	1	0	0	0	byte 1
1	1	1	1	0	byte 2
0	0	1	1	0	byte 3
1	1	1	0	1	byte 4
0	0	1	0	1	byte 5
0	1	1	0	1	byte 6
0	1	1	0	0	check byte 7

The unidirectional (0 \rightarrow 1) errors affect three adjacent lines (3,2,1) as shown, and impose exactly six changes. Now we get $N=1001$ and $R=0111110$. This is *exactly* the same condition as in Example 1, and "line 2 stuck on one" hypothesis is validated, since bytes 1 through 6 contain ones in line 2. Setting those six bits to zero will cause a mis-correction.

9. Conclusions

It is concluded that the two-dimensional codes are very nearly 100% (except in the cases of ambiguity as illustrated in Example 2) single-line correcting, and full 100% double-adjacent-line detecting codes with respect to unidirectional errors. The probability of mis-correction in the case of three-adjacent-line unidirectional errors remains very low, since a very specific error pattern and original pattern of X must coincide to cause a mis-correction.

It has been shown that byte-serial arithmetic can be carried out with operands which are encoded in two-dimensional residue and inverse-residue codes. Two-dimensional encodings provide a very powerful error-detecting and a substantial error-correcting capability for byte-serial arithmetic. Promising application areas are systolic arrays, multiple-precision arithmetic, and high-speed array computing.

REFERENCES

[AVIZ 62] Avizienis, A. "On a Flexible Implementation of Digital Computer Arithmetic," *Information Processing 1962*, C.M. Popplewell, ed., North Holland Publishing Co., Amsterdam, 1963, pp. 664-670.

[AVIZ 70] Avizienis, A., Tung, C., "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," *IEEE Trans. on Computers*, C-19: 733-745, August 1970.

[AVIZ71a] Avizienis, A. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Trans. on Computers*, C-20: 1322-1331, November 1971.

[AVIZ71b] Avizienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. on Computers*, C-20: 1312-1321, November 1971. Reprinted in *Best Computer Papers of 1971*, L. Petrocelli, ed., Auerbach Publishers, 1972, pp. 165-185.

[AVIZ 73] Avizienis, A. "Arithmetic Algorithms for Error-Coded Operands", *IEEE Trans. on Computers*, C-22: 567-572, June 1973.

[AVIZ 81] Avizienis, A., "Low-Cost Residue and Inverse Residue Error-detecting Codes for Signed-Digit Arithmetic," *Proc. 5th IEEE Symposium on Computer Arithmetic*, 1981, pp. 165-168.

[AVIZ 83] Avizienis, A. and C. S. Raghavendra, "Applications for Arithmetic Error Codes in Large, High-Performance Computers," *Proceedings, 6th IEEE Symposium on Computer Arithmetic*, 1983, pp. 169-173.

[BOSE 80] Bose, B., Rao, T.R.N., "Unidirectional Error Codes for Shift Register Memories", *Digest FTCS-10*, Kyoto, Japan, October 1980, pp. 26-28.

[GARN 58] Garner, H. "Generalized Parity Checking", *IRE Trans. EI. Computers*, EC-7: 207-213, September 1958.

[PARH 73] Parhami, B., Avizienis, A., "Application of Arithmetic Error Codes for Checking of Mass Memories," *Digest of the 1973 Int. Symposium on Fault-Tolerant Computing*, pp. 47-51, June 1973.

[PARH 78] Parhami, B., Avizienis, A., "Detection of Storage Errors in Mass Memories Using Low-Cost Arithmetic Codes," *IEEE Trans. on Computers*, C-27-4: 302-308, April 1978.

[TUNG 70] Tung, C., Avizienis, A., "Combinational Arithmetic Systems for the Approximation of Functions," *AFIPS Conf. Proc.* (1970 Spring Joint Computer Conference), 36: 95-107, 1970.

[USAS 78] Usas, A.M., "Checksum Versus Residue Codes for Multiple Error Detection," *Digest of the 8th Annual International Conf. on Fault-Tolerant Computing*, p. 224, 1978.

[WAKE 75] Wakerly, J. F., "Detection of Unidirectional Multiple Errors Using Low-Cost Arithmetic Codes", *IEEE Transactions on Computers*, C-24: 210-212, February 1975.

APPENDIX

Example 4: Line-Residue Checking

The byte-serial line-residue checking algorithm of Section 3 is illustrated below. The operand X is from Example 1, with the "line 2 stuck-on-one" error. Here $m=3$ and $k=8$.

check	0	0	1	0	0	0	0	1	
line 3	0	1	0	1	0	1	0	0	
line 2	1	1	1	1	1	1	1	1	
line 1	1	1	1	1	0	0	1	0	
line 0	0	0	0	1	0	1	1	0	
—————									
$\Sigma(L) = 010$	0	1	1	1	1	1	0	0	
$s_1 s_1 s_0 s_9$							$c_3 \rightarrow$		
							0	1 1 0	
$\Sigma(L)' = 010$	1	0	0	0	0	0	s_2	s_1	s_0

$\Sigma(L)' = \Sigma(L) + 2^3$, since $m=3$
Since $C_3=0$, $\Sigma(L)$ is selected as the check result, with
 $s_2 s_1 s_0 = 110$

ABSTRACT OF THE DISSERTATION

Layout from a Topological Description

by

Martine Denise Francoise Schlag

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Sheila A. Greibach, Co-Chair

Professor Milos D. Ercegovac, Co-Chair

The interconnection topology of a circuit does not, in general, correspond to a planar graph. However by encompassing the routing of a circuit in the specification, it is possible to obtain a planar characterization of the topology of a circuit. The planar topology of a circuit is formally defined and the use of specifications with planar topology for the layout of integrated circuits is examined. An applicative language(FP) is used to obtain circuit specifications with planar topology. The planar topology arises naturally out of the constructs used to specify the behavior of the circuit. An efficient mapping from planar topology to geometry is implemented. The problem of transforming the planar topology to minimize the interconnection complexity is addressed by exploiting the structural information of the specification as opposed to using only the planar topology.

A Proposal for the Systematic Design of Arrays for Matrix Computations

Jaime H. Moreno
Computer Science Department
University of California, Los Angeles

Report No. CSD-870019*
May 1987

11

*This research has been supported in part by the Office of Naval Research, Contract N00014-83-K-0493
"Specifications and Design Methodologies for High-Speed Fault-Tolerant Algorithms and Structures for
VLSI"

Abstract

We propose to develop a general and systematic methodology for the design of matrix solvers, based on the dependence graph of the algorithms. A fully-parallel graph is transformed to incorporate issues such as data broadcasting and synchronization, interconnection structure, I/O bandwidth, number and utilization of PEs, throughput, delay, and the capability to solve problems larger than the size of the array. The objective is to devise a methodology which handles and relates features of the algorithm and the implementation, in a unified manner. This methodology assists a designer in selecting transformations to an algorithm from a set of feasible ones, and in evaluating the resulting implementations.

This research is motivated by the lack of an adequate design methodology for matrix computations. Standard structures (systolic arrays) have been used for these implementations, but they might be non-optimal for a particular algorithm. Reported systems have used ad-hoc design approaches. Some design methodologies have been proposed, but they do not address many important issues.

A preliminary version of the proposed methodology has been applied to algorithms for matrix multiplication and LU-decomposition. The approach produces structures which correspond to proposed systolic arrays for these computations, as well as structures which exhibit better efficiency than those arrays. The results show that different transformations on a graph may lead to entirely different computing structures. The selection of an adequate transformation is thus directed by the specific restrictions and performance objectives imposed on the implementation. The designer can identify and manipulate the parameters that are more relevant to a given application.

END

9-87

DTIC